National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

University of Alberta

A Communication Model for the Software Systems Development Process - a Unifying Approach

©

by

Bogumila Kwiatkowska

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall 1991

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-70068-8

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Bogumila Kwiatkowska
TITLE OF THESIS: A Communication Model for the Software Systems
Development Process – a Unifying Approach

DEGREE: Master of Science
YEAR THIS DEGREE GRANTED: 1991

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis not extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) B. Kwiatkowska . . . .

Permanent Address:
6802-110 Street,
Edmonton, Alberta

Date: October 9, 1991

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **A Communication Model for the Software Systems Development Process – a Unifying Approach** submitted by **Bogumila Kwiatkowska** in partial fulfillment of the requirements for the degree of Masters of Science.

H. James Hoover
Dr. H. J. Hoover (Supervisor)

. . . .

. . . .

PG Sorenson
Dr. P. G. Sorenson

Duane Szafron
Dr. D. Szafron

Dr. D. G. Fisher

Date: . . . .

To my husband Andrew and my parents.

Abstract

This thesis concerns itself with modelling the software systems development process. We propose a model that focuses on the two most important, yet neglected, aspects of software development: *communication* and *people*. We demonstrate that communication (both formal and informal) between project participants is the most important factor in the systems development process and that the quality of software systems depends on the effectiveness of the underlying communication system.

A communication-based description provides a realistic and a uniform model for all development activities. In fact, more than 50 % of software professionals' time is spent communicating with other project participants. This communication includes such complex processes as negotiating, learning, and managing. However, none of these processes is fully described in existing models.

Furthermore, the communication model provides a uniform, high level description for the integration of different methods and techniques. Because of its generality (communication uniformly describes all activities in all types of software projects), it allows for the specification of existing models, for example, waterfall, spiral, transformational, rapid prototyping, or object-oriented.

The communication model describes the software systems development process as a network of agents exchanging messages over a set of channels. In this paradigm, all activities (including the initial development and post-implementation) are represented as communication processes providing specific services.

Since the communication model is *generic*, it can be implemented in specific projects using various methods and techniques. In our work, we describe a *state* and *transition* based formalism as an implementation example. This formalism, *Communicating Abstract Machines*, is based on (1) the Communicating Finite State Machines, (2) formal specification language for communication protocols (Estelle), and (3) structural decomposition techniques used in Statecharts. Processes represented by Communicating Abstract Machines are easily visualized and can be executed by both people and machines.

Acknowledgements

With sincere gratitude, I wish to thank my supervisor Dr. James Hoover for his invaluable guidance and encouragement throughout the research.

Contents

1 Introduction 1
1.1 A New Approach to Software Systems Development 4
1.1.1 Communication Model Proposal 6
1.2 Formal Communication in the Development Process 7
1.2.1 Oral Communication 7
1.2.2 Written Communication 9
1.3 Group Communication 10
1.3.1 Inter-Group Communication 11
1.3.2 Intra-Group Communication 13
1.4 Types of Communication Processes 13
1.4.1 Negotiation Process in Systems Development 14
1.4.2 Learning Process in Systems Development 14
1.4.3 Managing Process in Systems Development 15
1.5 Practical Aspects of Communication Model 15
1.5.1 Time Spent on Communication 15
1.5.2 The Number of Communication Channels 16
1.5.3 Examples of Project Failures due to the Communication Problems 16
1.5.4 Current Practices in Software Projects 17
1.6 Communication and the Organizational Structure 17
1.6.1 Communication Principle 18
1.7 Outline of this Thesis 18

2 Models for the Software Systems Development Process 20
2.1 Software Engineering and Systems Analysis and Design 20
2.2 Methods used in Software Engineering and Systems Analysis and Design 22
2.3 Models for the Software Systems Development Process 24
2.3.1 Waterfall Model 24
2.3.2 Spiral Model 24
2.3.3 Criticism of the Software Life Cycle 25

2.3.4 Transformational Method 25
2.3.5 Object-Oriented Method 25
2.3.6 Parallel Software Development Method 26
2.4 Software Development Techniques 26
2.4.1 Rapid Prototyping 26
2.5 A Process Description of the Software Systems Development Process 27
2.5.1 Software Process 28
2.5.2 Software Process Programming 28
2.6 Communication Model – a new Paradigm 30
2.7 Communication – a Unifying Approach 32
2.7.1 Related Fields 33
2.7.2 The Communication Model and the Groupware 33
2.7.3 The Communication Model and Organizational Communication 35
2.8 A Systematic Approach to Modelling 36
2.8.1 Step 1 - Software Systems Development Process Definition . 39
2.8.2 Step 2 - Why the Software Systems Development Process should be modelled? 39

3 Software Systems Development Process Modelling 40
3.1 Step 3 – Software Systems Development Process Redefined 40
3.2 Step 4 – Software Systems Development Process and its Model . . . 40
3.2.1 The Characteristics of the Software Systems Development Process 42
3.2.2 Components of the Development Process 44
3.3 Step 5 - A Rigorous Model for the Software Systems Development Process 45
3.4 Step 6 - Formal Specification 50
3.4.1 Organizational Advantages of the Communication Model . . 51
3.4.2 Formal Model and an Automated Task Verification 52
3.5 Step 8 - Implementation of the Communication Model 52

4 A Rigorous Model For The Software Systems Development Process 54
4.1 A Project Definition 55
4.2 Participants 56
4.2.1 Participant Definition 57
4.3 Tasks 57
4.3.1 Task Specification 58
4.3.2 Task State 59
4.3.3 Task History 59
4.3.4 Control Tasks 60

4.4 Objects 60
4.4.1 Modules 60
4.4.2 Knowledge Repositories 60
4.4.3 Project Resources 61
4.5 A Communication Based Description for the Waterfall Model 61
4.5.1 A Communication Based Description for the Spiral Model . 65
4.5.2 A Communication Based Description for the Prototyping Approach 67

5 Modelling of the Systems Development Activities. A Case Study 69
5.1 Systems Development Activities 69
5.1.1 The Communication Model Representation for the Development Activities 70
5.1.2 Structured and Unstructured Processes 70
5.2 Unstructured Task. A Case Study 72
5.2.1 Process Specification 73
5.2.2 Task 1. Survey of the Current Situation 74
5.2.3 Task 2. Analysis of the Survey Reports 75

6 Tasks Specification. A Case Study 76
6.1 Communication 76
6.1.1 Asynchronous Communication 77
6.1.2 Three Addressing Schemas 77
6.1.3 Port-to-Port Addressing Method 78
6.1.4 Client/Server Paradigm 78
6.2 Message Definition 78
6.2.1 Message Related Data 78
6.2.2 Task Related Data 79
6.2.3 Case Study. Messages for Task 1 80
6.3 Meta-Task Operations 81

7 Formal Specification of Systems 86
7.1 Levels and Domains of Specification 86
7.2 Characteristics of the Specification 87
7.2.1 Abstract and Linguistic Nature of Specification 87
7.2.2 Dynamic Nature of Specification 87
7.2.3 Complex Nature of Specification 88
7.3 Intertwining of the System Specification and Implementation 88
7.4 Specification Definition 89
7.5 Formal Specification 90
7.6 Verification and Validation 90

8 Communicating Abstract Machines as a Formal Specification for the Communication Model 91
- 8.1 Communication System Modelling 91
- 8.2 Simple Finite State Machine 94
  - 8.2.1 Deterministic Finite State Machine Definition 95
  - 8.2.2 Nondeterministic Finite State Machine 96
- 8.3 Abstract Machine (Extended Finite State Machine) 96
- 8.4 Communicating Finite State Machines 97
  - 8.4.1 Extended State Transition Language (Estelle) 98
- 8.5 Representation Methods for FSM 99
- 8.6 Communicating Abstract Machines 100
- 8.7 Specification Example. Subtask 5 102
- 8.8 Time Representation and its Functions in the Communication Protocol Specification 104
  - 8.8.1 Formal Time Specification 104
  - 8.8,2 Example. Timing Specification for Subtask 5 107

9 Conclusions 110
- 9.1 Contribution of this thesis 110
- 9.2 Future Work 111

A Implementation Examples for the Communication Model 113
- A.1 Participant Implementation 113
  - A.1.1 Resources Assigning Procedure 114
  - A.1.2 Human Resources Assigning Procedure 114
  - A.1.3 Human Resources Assigning Example 115
- A.2 Services, Mechanisms, and Facilities 116
  - A.2.1 Services 117
  - A.2.2 The Advisory Mechanism Example 117
  - A.2.3 Reporting Facilities 118

List of Figures

1 Communication in the Systems Development Process 11

2 Evolution of Models for Software Systems Development Process . . 31
3 The Communication Model and other Fields 33
4 Modelling of the Software Systems Development Process 38

5 Characteristics of the Software Systems Development Process . . . 42
6 Mapping between the Development Process and its Model 44
7 Three Aspects of the Communication Model 46
8 Vertical Expansion of the Process Specification 47
9 Horizontal Expansion of the Process Specification 47

10 A Task Life Cycle 59
11 Communication Model for the Systems Analysis 62
12 Knowledge and Modules Repositories 64
13 A Cycle in the Spiral Model 65
14 A Communication Based Representation for the Spiral Model . . . 66
15 A Communication Based Representation for the Prototyping Technique 67
16 A Communication Based Representation for the Training Process . 68

17 Systems Development Activities 69
18 Control of the Structured Task Execution 71
19 Control of the Unstructured Task Execution 71
20 Client Service Improvement Task 74
21 Survey Task Specification 74
22 The Relationships between the Survey Reports 75

23 Meta-task Operations 82

24 Limitations of simple FSMs and their Solutions 94
25 Functional Decomposition of Subtask 5 102
26 Communication Abstract Machine describing the behaviour of MG 103
27 Communicating Abstract Machine describing the behaviour of PM . 103

28 A Communicating Abstract Machine with Time-outs 107

29 Structure of Services and Mechanisms 118

Chapter 1

Introduction

> The structure of a software product is isomorphic to the structure of the project which developed it. (Conway's Law [Conway 68] [Boehm 81])

Despite many advances in computer technology and computing science, the "software crisis", identified in the 1960s, has not disappeared in the last thirty years. Software projects notoriously overrun their budgets, pass their deadlines, and, most importantly, fail to meet users' requirements. Final products are often unreliable, not well documented, and difficult to change. The current software development methodologies, techniques, and computer-aided tools alleviate some problems, but they do not address the fundamental issues of the software systems development process. The software crisis results from the absence of adequate models and a lack of understanding regarding the nature of software systems and the peculiarity of the development process.

In terms of human history, software systems are a new phenomenon. The development of such systems, whether called *software engineering*, *programming-in-the-large*, *programming-in-the-many* [Fernstrom 89], or *systems analysis and design*, is a very young discipline. Yet this discipline is growing exceptionally quickly and it has begun to affect almost every single aspect of human life. As has happened with many new branches of knowledge, the development of software systems has been modelled on other well established and somewhat similar fields. These are *engineering*, *formal logic*, *office work*, and *art*. Each of these approaches captures certain aspects of software systems, but none of them represents sufficiently the entire complexity of these systems and their development methods. Unfortunately, all four approaches have also introduced many misconceptions, and they are purveyors of two extreme perspectives, claiming that the software system belongs either to the *art* or the *engineering* domain. This antagonistic "either-or" view has caused a lengthy battle between the advocates of the engineering and art approach.

Engineering Approach

The *engineering* approach regards software systems as physical products and the development process as an industrial assembly line production. According to this model, a software system is designed, prototyped, constructed, and distributed; it then requires only a little maintenance to repair or replace the worn out parts. However, this analogy does not apply. Firstly, a software system is not a physical object, at least not in the same sense as a car or a house. Software systems can exist in many different forms, as for example magnetic fields, electric pulses, or even holes in a paper. Furthermore, software can be distributed over several physical locations (it is difficult to imagine a distributed car which still functions normally). Secondly, software systems do not wear out from use; therefore, they do not require maintenance. What is, however, expected from a software system is its ability to adapt to changing requirements and operate in evolving environments. Software systems must be expandable, reducible (able to be scaled down), and flexible enough to accommodate frequent changes. Software systems can not be modelled as physical products; they are dynamic systems providing complex services to people. The engineering approach also models two valid aspects regarding the development of software systems: (1) software systems are created by groups of people and (2) the creation process must be organized.

Observation 1

Software systems are not physical objects in the same sense as a car or a house. They can be distributed and transformed from one physical form into another, and, furthermore, they do not wear out through use. They provide *services* to people, and since people and their environments change, services must, accordingly, be modelled by dynamic systems. On the other hand, the creation of software systems involves a large group of people and requires good organization.

Formal Logic Approach

Since computers are based on logic, the *formal logic* approach to software systems is justifiable. A strict mathematical approach looks at the development process as a transformation from formally defined specification into a corresponding implementation. This approach provides several good specification and verification methods; however, it also has its difficulties. The problem lies in the formal specification of human activities. Since software systems model human reality and a changeable environment, a formal description of these phenomena is, at best, tedious, if not impossible. Attempts to define formally human language have shown that this task is not easily accomplished. Therefore, taking into consideration the fact that services required by people change continuously, an entirely formalized specification of a large system is not feasible. Yet, the formal logic approach is useful in specification and verification of the high risk systems, for example, a nuclear power plant control system. The formal logic approach has one further difficulty: the transformation and verification procedures are only as reliable as the supporting proof theory.

Observation 2

The development of software systems should be based on formal techniques. However, a fully formalized development process for a complex system is not feasible.

Office Work Approach

The *office work* approach perceives a software system as a collection of documents and the development process itself as a production of code or specification lines. This tradition dates back to the key-stroke entry methods when everything intended for the computer was typed. The predominant typing activities and vast amount of printed paper fostered a resemblance to an office environment. However, software systems can not be defined only in terms of produced documents. Software systems require problem finding, creative thinking, and problem solving techniques ([Weinberg 71], [Tracz 79], [Naur 85]). It is irrelevant whether the programs are typed, copied and pasted, scanned, or digitized and stored as a picture. It is important, however, that the problems are defined and solved . On the other hand, the office work approach has introduced another crucial aspect of the software development – *traceability*, which means that the software development must be monitored while all tasks and their results must be traceable.

Observation 3

The creation of software systems involves many problem finding and problem solving techniques. These techniques should be traceable.

Art Approach

The *artistic* approach to the development of software systems is still popular among many developers. *Artistic*, in this context, means that the system development should be based on intuition and experience rather than on rules or formal definitions. However, software systems are not art works. They are not produced to create an impression or feeling but to solve a problem with stated requirements. They are created by cooperating groups of people using limited resources, and they are required to provide strictly specified services.

The one major similarity between a literary work and a software system is that they are both communication systems. However, literary works involve human-to-human communication, but software systems involve two types of communication: human-to-human and human-computer. Therefore, software systems must use languages which are understood by both humans and computers.

Observation 4

Software systems are based on both human-to-human and computer-human communication. As a result, they use specification languages understood by humans and computers.

1.1 A New Approach to Software Systems Development

None of the four approaches: engineering, formal logic, office work, or art provides an adequate model for the software systems development process. Although they model some aspects of the development process, they do not capture the communication processes underlying software systems and their development.

A valid model for the software systems development process must consider two realms: (1) modelling of the software systems and (2) modelling of the development of software systems. These two aspects are interrelated, and they can not be treated separately ([Conway 68], [Daly 79], [Tully 84]). Thus, the model for the development process depends on the underlying assumptions about the nature of a software system. For example, the false assumption that a software system is similar to a physical object has led to the concept of an assembly-line model for *software production*.

Moreover, modelling of software systems and their development must be based on three assumptions: (1) software systems provide *services*; (2) software systems are created and used by *people*; (3) Software systems are themselves *communication systems*, and they are created in a *communication* process.

Software Systems as Services

Software systems should be modelled as *services*, which correspond to changing human needs. People operate in changing social, political, economic, and natural environments. Moreover, people themselves undergo constant changes. Thus, both software systems and their development process must be designed as dynamic, flexible, and expandable processes.

The *service* oriented model is concerned with the entire development process – specification of users' requirements, construction (acquisition) of the tools to provide desired services, and the verification of the tools against their specifications. The word *development* is understood here in a broad sense as an ongoing process, encompassing the entire life of a system. Thus, the service-based model does not distinguish between the traditional pre- and post-implementation phases. Instead, it has three components: specification, construction, and verification, all of which repeat throughout the entire life of the system.

Software Systems Development as a Human Activity

The software systems development process should be modelled as a *human* activity, specifically as a group activity. Software systems are designed, studied, modified, and used by people. Even the best designed and formally specified system can never be used if it does not address the particular users' needs at a particular time. A *Human* centered development process requires an explicit modelling of the *participants*: end-users, systems analysts, programmers, technical support staff, management, project sponsors, standardization groups, quality assurance groups, and technical writers. It is paradoxical that while most of the textbooks for systems analysis and design and software engineering ([Martin 91], [Macro 90], [Powers et al. 90], [Flaatten et al. 89], [Whitten et al. 89], [Senn 89], [Amadio 89], [Yourdan 89], [Kozar 88]) advocate the users' involvement in the entire development process, none of them uses a model which includes an explicit description of the participants. This fact is even more startling in light of the empirical studies which have shown that the end-users' and managers' satisfaction (the feeling of system ownership, the positive and open approach to the computerized system, and the extent of system use) is positively correlated with their participation in the system development. A good example is Montazemi's study of 83 small firms [Montazemi 88]. Applying explicitly stated measurements and statistical calculations, he arrives at the conclusion that users' participation is one of the major factors in successful software development.

The users are involved in all development activities: requirements specification, system analysis review, system design prototyping, acceptance testing, and system evaluation. They also participate in system modelling, feasibility studies, hardware and software acquisition, and training. Nevertheless, the currently existing software development methods do not model user roles; in fact, they do not describe any participants. Thus, a realistic model of the development process must include explicit representation for the roles played by all participants.

Moreover, software systems are not created by personal endeavor nor in groups whose members do not communicate. On the contrary, software systems are created and used by people who constantly communicate. Therefore, the development process must be modelled as a human activity as well as a communications activity.

These two aspects are complementary to each other — *human* and specifically *group* activity requires communication, while, at the same time, communication requires human participation.

Software Systems Development as a Communication Process

Effective *communication* is the single most important factor in all software development activities [Powers et al. 90]. Without good communication between project participants: users and developers, developers and sponsors, technical support groups and systems analysts, developers and vendors; it is not possible to specify user requirements or build and acquire tools to provide specified services ([Bostrom 89], [Bostrom 88], [Curtis et al. 87], [Guinan and Bostrom 86], [Bostrom 84]). This fact is particularly evident in large projects or systems distributed among various organizations or geographical locations. It is even more apparent in the development of non-traditional (non-transaction based) systems such as decision support systems, communication support systems, management information systems, and knowledge based systems. The requirements specifications for these systems depend essentially on effective communication between the system developers and the users.

Consequently, explicit representation for *participants* and *communication* are the two most important components of a valid model for the software systems development process. In fact, each development activity is a communication activity, and the development process is itself a complex communication system. Communication is also the common denominator of software systems, development activities, and management processes.

1.1.1 Communication Model Proposal

We propose a *communication model* as a *unifying* and representative method of description for the software systems development process. We use the word *unifying* in reference to three aspects of the software systems creation: (1) communication can be used to model both systems and their development process; (2) communication-based models describe both the specification of the development process and the management of the actual development process; and (3) communication integrates the existing software development methods.

1.2 Formal Communication in the Development Process

Large development projects follow in fact well specified communication procedures (protocols). They often use rigorous communication procedures and exchange a vast number of documents to ensure that the design decisions are approved and authorized. A large part of this communication follows a project organization chart and is executed using a well defined scenario: the appropriate documents are filled out, delivered, signed-off, and stored. This documented and well organized communication is often called *formal communication* and will be used in this sense in our model.

We define *formal communication* as a project related technical communication that is explicitly described and has documented results. It can result in a change in the specification or task execution. It can also provide affirmation in a negotiation process or information in a learning process. Formal communication is therefore goal-oriented and has a particular plan (scenario) to achieve that goal. The word *formal* does not refer to human communication style. In fact, our formally specified communication tasks can be executed in an *informal* way. Consequently, our model describes formal communication, but, at the same time, it does not preclude or obstruct the informal interaction among participants. Informal interaction should be encouraged as a natural human need for social contacts and creative information exchange [Weinberg 71]. It is well known that many crucial decisions and design ideas are the result of an informal interaction, communication initiated freely by the participants without formally specified goals and results. Informal communication can be continued within the development system using electronic mail and computer-supported conferences or it can be realized in face-to-face meetings. However, the results of such communication should be documented to preserve the integrity of the system.

Formal communication has two basic forms: *written* and *oral* ([Constantine 90], [Rettig 90]). Oral communication incorporates many forms, for example, small group discussions, phone calls, meetings, interviews, presentations, walkthroughs, and training sessions. Written communication includes memos, status reports, meeting minutes, bulletins, analysis and design documents, end-users' and training manuals.

1.2.1 Oral Communication

Meetings

In well-structured organizations, meetings are carefully scheduled and prepared. A meeting agenda is prepared in advance and distributed to all participants. It states the purpose of the meeting and provides a timetable for questions and discussion. Required documents are attached to the meeting agenda, and they are reviewed by the future meeting participants (or at least one designated reviewer). The meeting itself follows the agenda and is documented by a designated participant, the recording secretary. Some meetings require *facilitators*; some also have designated critics (devil's advocates). After the meeting, the minutes are published and distributed among participants and interested parties.

Walkthroughs (technical reviews) are a good example of highly structured and goal-oriented meetings, designed to perform quality review and identify possible errors. Team walkthroughs, performed accordingly with predefined scenarios, provide better and faster results. An example of empirical study is the Electronic Meeting System (EMS), GroupSystems Concept, developed at the University of Arizona and used by over 33 of IBM's sites. ([Grohowski et al. 90]). In this experiment, the estimated time (provided by expert developers) was compared with the actual time spent on meetings using the electronic meeting system. The average saving per session was 51 % ([Valacich et al. 91], [Grohowski et al. 90]).

In general, the rigorously described and executed meetings are shorter and more "subject oriented"; they also save participants' time since they do not require physical presence at the same location. Furthermore, they allow off-line (asynchronous) communication.

Structured Interviews

An interview with the user is the basic tool used in systems analysis [Saunders 91]. In a structured interview, the interviewer learns about the interviewee (position, character, biases) and prepares a list of questions organized into a scenario (guide). The interview is always scheduled. Often, the interviewee is provided with the set of questions (problem areas) before the interview. The interview itself adheres to the scenario. The interviewer records the results by making written notes (if the situation permits). After an interview, the significant points are documented. Sometimes the post-interview documents have to be authorized.

Research in communication demonstrates that well structured interviews with clients drastically increase the accuracy of requirements specification and shorten the required time. Bostrom [Bostrom 89] observed a significant improvement in requirements quality, after training the users and the developers in effective communication. All project participants were taught the PRECISION model. This model was originally developed by neuro-linguists for therapeutic purposes and later adapted for the business setting.

Group Work Sessions

The traditional method of systems analysis requires a number of individual interviews with the end-users. However, in many cases several individual interviews can be replaced by one group meeting. The group work sessions are conducted by systems analysts with a group of users (for example: end-users, management, and project sponsors). By employing this technique, many conflicting opinions can be resolved during one meeting. The best known group session technique is IBM's Joint Application Design (JAD) [Andrews 91].

1.2.2 Written Communication

Written communication incorporates many functionally different documents: proposals, service requests, feasibility studies, requirements statements, design and implementation specification, system and program-level documentation, user manuals, change requests, conversion specifications, backup and recovery procedures, project histories, problem logs, status reports, memos, and test plans. These documents can be divided into two classes: *internal* and *external*. Internal documents are developer oriented (technical documents) or user oriented (management oriented and end-user oriented documents). External documents are prepared for standardization, education (case studies), legal, and commercial purposes.

Written communication is important for a number of reasons: (1) it creates a permanent recording which can be analyzed, organized, and studied "off-line"; (2) it provides a "parallel" medium as opposed to sequential oral communication; (3) it uses a written form of language which is far more precise and structured than the spoken form; and (4) it involves visual media (graphs, charts, drawings).

In practice, system documentation is used for asynchronous communication between project participants. Since the initial development of a large project takes from one to three years, many participants never have a chance to communicate in person and have to rely exclusively on written documentation [Curtis et al. 88].

Developer/user communication extends over time (in some existing systems over twenty years). Users can perform specific functions long after the system has been developed and all initial developers have left. Therefore, end-user documentation and training manuals are the most important documents for presenting and explaining the system to its users.

Singer in his book on written communication for software developers describes the following rules [Singer 85]:

1. The number of problems in a computer system is inversely proportional to the quality of the written specification.
2. The greater the number of undocumented changes to a system, the higher the

probability that those changes will be wrong, or that no one will understand the changes even if they are correct.

3. The more a company relies on meetings and discussions to settle issues without documenting the results, the more misunderstanding will occur.

The importance of an accurate documentation can not be overstressed. Software systems are major financial investments and the absence of valid documentation translates into major financial losses. Attempts to "recreate" specification from the programs (reverse engineering) can be beneficial for additional verification and fine tuning of the system. However, reverse engineering does not replace specification.

Histories of all development activities can be used not only for an internal analysis and audit but also for external comparisons and education. A well kept history of the entire development process can serve as valuable educational material (a case study). Retrospective analysis of successes and failures of complex software systems will significantly improve the research in software engineering. Therefore, it is unfortunate that million-dollar "bugs" are not well described, analyzed, and published for, at least, educational purposes. In many cases, politics preclude publication of any details. However, a growing number of researchers and practitioners advocate recording and publishing of the lists of software problems ([Weinberg 83], [Rettig 90], [Neumann 91]).

The following excerpt from Jerry Weinberg's column [Weinberg 83] is an illustration for Singer's second rule as well as for the high cost of inadequate documentation:

> I keep a confidential list of the world's most expensive programming errors. All of the top ten on the current list are maintenance errors. The top three cost their organization $ 1,600,000,000, $900,000,000, and $245,000,000 — and each one involved the change of a single digit in a previously correct program. In all three cases, the change was "so trivial" it was instituted casually. A supervisor told a low-level maintenance programmer to change that digit without any written instruction, no test plan or nobody to read over the change.

1.3 Group Communication

Systems development process requires ongoing and extensive *group communication*, which involves a large number of participants, using different languages and communication patterns (culture, age, and individual differences) ([Kettelhut 91], [Bostrom 89]). Thus, the *negotiation*, *learning*, and *managing* processes must be analyzed in conjunction with group dynamics. In general, communication between project participants can be classified as *inter-group* and *intra-group*. The following two subsections describe the issues related to (1) communication between groups of participants and (2) communication between participants inside a heterogeneous group.

1.3.1 Inter-Group Communication

This subsection provides an example of communication channels and their functions in the software systems development process. Figure 1 illustrates an example of the communication between groups of participants. The users, managers, and project sponsors are categorized as a class of participants using an application domain. Systems analysts, trainers, and technical writers are associated with the information (conceptual) model; the application programmers and technical support group are associated with a particular implementation of the designed system. The other three groups: vendors, standard enforcement groups, and legal advisors are involved in commercial (financial), organizational, and legal aspects of the system.

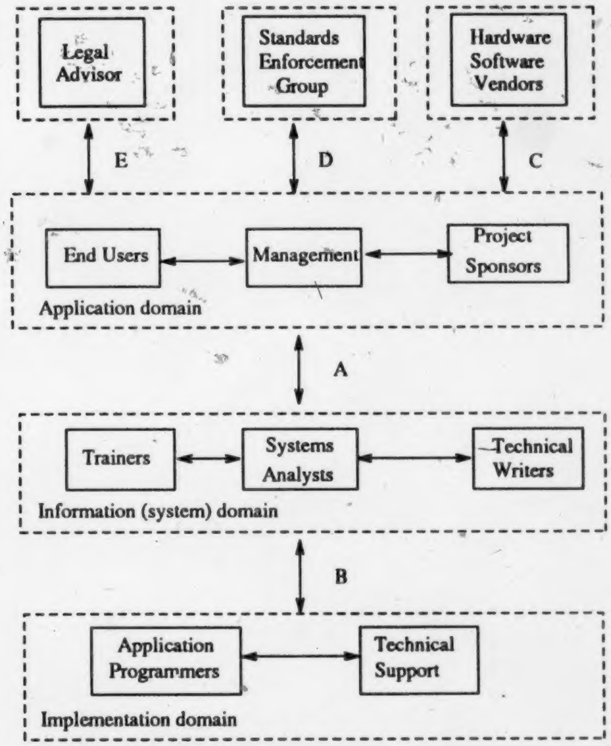
Legal Advisor
Standards Enforcement Group
Hardware Software Vendors
E
D
C
End Users
Management
Project Sponsors
Application domain
A
Trainers
Systems Analysts
Technical Writers
Information (system) domain
B
Application Programmers
Technical Support
Implementation domain

Figure 1: Communication in the Systems Development Process

Communication between the Users and the Systems Analysts

The most important communication takes place between the future users of the system and its developers (channel A in Figure 1). The result of this complex and extensive communication is the users' requirements specification.

In practice, users communicate with a specialized group of developers – systems analysts. This group serves as a mediator between the users and the technical group of developers. The user group and the systems analyst group together create an *application domain based* model and an *information domain based* model. However, this process encounters two communication problems:

1. The user group and the systems analyst group have different backgrounds, use different languages, and function in different organizational subcultures [Guinan and Bostrom 86].
2. The user group is often heterogeneous. It can include many different individuals or departments with conflicting requirements. The systems analyst group can also be heterogeneous, with members varying in background and experience.

The problems in user/analyst interactions can be a major source of specification errors. As it is noted by Salaway in [Salaway 87], these two groups must not only communicate, but must also acquire knowledge from at least three domains: application (problem), information system (solution), and implementation (software and hardware). With the growing number of computer literate users and simple to use analysis and design tools (CASE tools), communication between these two groups is becoming easier. However, for the majority of complex tasks the systems analyst remains the central communication channel.

Communication between Systems Analysts and Technical Specialists

Systems analysts, application programmers, and the technical support group have two perspectives on the same system (channel B in Figure 1). They use different languages and operate in dissimilar environments. The mapping from the conceptual model of the system to the physical model is done through communication between these three groups. Formal specification techniques provide some tools to automate the physical implementation process (code generators). However, a complex project still requires considerable human participation and extensive communication.

Communication with the Software and Hardware Vendors

Since many components of large systems are built from the ready made software products (with some necessary adjustments), the software vendors are important project participants. Efficient communication between the user representative group (acquisition committee) and the vendors (channel C in Figure 1) is critical to a project's success. The product acquisition task involves gathering and processing information about available hardware and software products, their cost, and warranty policies. It may also require negotiation with the vendor as well as legal advice.

Communication with the Standard Enforcement Groups

Usually, quality assurance is conducted by an external organization — a third party, independent from the users and the system developers. However, the quality assurance or certification group must communicate with project management and, often, also with the systems analysts (channel D in Figure 1).

Verification and certification procedures depend on the type of enforced standards. Generally, standards can be classified into two groups: application oriented standards and system development standards.

Application oriented standards are the particular industrial policies, for example: standards for communication protocols. Industrial standards are usually verified by independent agencies and government committees. For example, the computer networks protocols are certified by the International Standards Organization (ISO) [Tanenbaum 88].

The analysis, design, and implementation oriented standards are the methodologies, methods, models, and techniques accepted as obligatory by specific projects or entire organizations. These standards are verified by a special group of participants or by an external quality assurance group (Certification Agency).

1.3.2 Intra-Group Communication

It should be emphasized that the majority of groups are themselves heterogeneous. The user groups (end-users, management, project sponsors) are often incongruous and their requirements are incompatible. The development teams often consist of a multiplicity of private consulting companies, individual contractors, and government employees. Since participants belong to different groups (many groups at the same time) their loyalties can be divided [Umstot 87]. The conflicting interests of the participants and their cross-cultural differences can only be consolidated by clearly and precisely specified tasks.

1.4 Types of Communication Processes

Communication is the basis for three types of processes: *negotiation*, *learning*, and *managing*.

1.4.1 Negotiation Process in Systems Development

The process of negotiation is the most important activity in the development of the software systems. It is a sequence of decisions to accommodate the most user requirements using limited budget and in limited time. Since systems are created by large and often diversified groups of people for other large groups, the specification of requirements and the mapping into the information model requires many compromises. Whereas negotiation has been studied by disciplines such as management [Boehm and Ross 89], psychology [Umstot 87], and linguistics (discourse analysis) [Stubbs 83]; little has been done in studying negotiation in the context of a system development. Recently, Robinson [Robinson 90] presented studies done on negotiation behaviour during requirements specification. Robinson pointed out that the negotiations, their results, and the rationalization behind the decisions should be fully recorded. Thus when the particular specification must be changed, the records (histories) of previous negotiations and decisions can be analyzed.

Negotiation and conflict situations exist not only in the specification phase, they are a natural consequence of the diversity of groups and people participating in the system development. The following examples illustrates a typical situation in the contract based systems, when most private companies are natural competitors. The situation involves two private companies *A* and *B* with conflicting interests. Company *A* is handing over a project to company *B*. However, if company *B* appears to be incapable of taking over the project, company *A* will stay and company *B* will lose the project. Obviously, these companies have conflicting interests and their employees are not willing to cooperate. The division of work between company *A* and *B* must always be negotiated and fully documented. Therefore, effective task executions can be achieved only by rigorously specified communication protocols imposed by an independent group.

1.4.2 Learning Process in Systems Development

The term *learning* is used here in a very broad sense – as a process involving acquisition and dissemination of knowledge or training processes (acquisition of skills). The software systems development process brings together many groups of people with very different backgrounds.

Understanding and communication between these groups are difficult since each group uses a different model of the system: *application domain* model, *information domain* model, and *implementation* model [Wand and Weber 89]. These three models are illustrated in Figure 1.

The first model is built by application specialists (for example: specialists from medicine, accounting, or physics), with the help of system specialists (generalists), in the process of requirements specification.

The second model is constructed by mapping from the application model to the information domain. It involves both application specialists and systems analysts. However, it is the systems analysts (information specialists) domain.

The last model is built by the technical group (application programmers and technical support group) by mapping the system model (design specification) into a particular physical system. Thus, the implementation model is realized as an artifact — a collection of programs and related documents.

Each mapping requires knowledge from more than one domain. Therefore, the domain specific knowledge must be transferred between groups. This transfer requires both knowledge description (capturing) and knowledge acquisition (learning).

Learning is one of the most important and most time consuming activities ([Curtis et al. 87], [Koubek et al. 89]). Curtis expresses this in the following statement: "Much of what occurs during design is not designing, but the learning required in order to design successfully." However, as important and as time and resource consuming as it is, learning is excluded from the existing systems development models.

1.4.3 Managing Process in Systems Development

The managing process involves *planning*, *scheduling*, *staffing*, and *controlling* of the development activities. It requires a well structured and well defined communication network and fast access to vast amounts of information.

Development activities are executed within a specified time frame while using limited resources. Thus, efficient management and effective resource utilization are important in deciding about the success of software projects.

1.5 Practical Aspects of Communication Model

1.5.1 Time Spent on Communication

At least half of a projects' development time is spent on communication. Studies of programmers and other software professionals show that they spend more than 50 % of their time on job related communication. Bell Laboratories' study of 70 programmers [Boehm 81] indicated that 32 % of the programmer's time is spent on job related communication, 6 % on training, 16 % on reading, while only 13 % is spent on writing programs. IBM User's Study Group [McCue 78] reported that 30 % of programmer time is spent working alone and 70 % is spent working in groups. Sullivan [Sullivan 88] studied software professionals and reported that the amount of time spent on human interactions ranged from 32 % to 93 % with a mean of 56 %. Grohowski, McGoff, Vogel, Martz, and Nunamaker [Grohowski et al. 90] stated that business managers spend from 35 % to 70 % of their time in meetings – these numbers do not include communication outside the meetings.

1.5.2 The Number of Communication Channels

The number of communication channels is proportional to the number of project participants. Theoretically, each participant can communicate with all other participants. Assuming that the class of participants includes practically everybody who in some way (for some time) is involved in the development process (project sponsors, end-users, management, systems analysts, programmers, technical support, standard enforcement group, system auditors, independent consultants, hardware/software vendors, technical writers, trainers), the number of participants, even for small projects, is considerable. A project with just ten participants has ninety one-way channels.

Our communication model provides a tool to describe, analyze, and modify communication patterns. In this process, unnecessary communication channels can be eliminated and communication bottlenecks can be identified.

1.5.3 Examples of Project Failures due to the Communication Problems

Projects fail usually not because of technical difficulties but because of management and interpersonal problems. They fail in most cases due to ineffective communication. Interestingly, the first major failure is illustrated by the biblical story recording the unfinished construction of the Tower of Babel (Gen. 11:1-9). The project had sufficient resources and a great plan, but it failed due to a single problem – the builders had begun to speak different languages and communication between them was thus impossible. The history of humankind is filled with many historical events where miscommunication played the major role. The history of software system development is far shorter (at most 30 years); however, it has already recorded many system failures and mistakes due to communication problems. Many cases can be found in the computer literature; however, for obvious reasons they do not provide names or places. To illustrate the point, we will present a case described by Rettig [Rettig 90]. A telephone billing system had not been charging new customers for several months. The program, which calculated installation charges, had not been installed. Since, obviously, the customers did not complain, the free installation was in place for several months. The problem was identified as a lack of documented procedure for the system installation (planning and coordination was by word of mouth) and a lack of communication (the technician, during installation, found that one program was missing, but the technician 'patched around' the problem without notifying anybody).

1.5.4 Current Practices in Software Projects

In practice, large projects use computer supported communication tools (electronic mail, electronic bulletin board – News, network user groups) for interactions between people. They also use some forms of communication specifications (organizational charts, company policies, informal guidelines for meetings, user interviews, and team walkthroughs) and document all major communication activities (memos, meeting minutes, walkthrough reports). Our communication model incorporates existing practices and provides tools for an explicit description of the communication procedures.

The existing technology, especially computer networks, support many communication oriented tools: electronic mail, interest groups, bulletin boards, computer supported conferences (groupware). These tools are often used, especially in academic and research environments (UNIX system). Andrew Message System (AMS), described by Borenstein [Borenstein 91], is an example of a large scale mail and bulletin board system. Many other communication tools (for example: Professional Office System PROFS) are used by governmental organizations and private companies. However, communication tools, though used in development related activities, are not formally described and are not included in the software development process models.

1.6 Communication and the Organizational Structure

It has been shown by many researchers and practitioners ([Conway 68], [Weinberg 71], [Boehm 81]) that there is a homomorphism from the structural organization of the system to the structural organization of its designers. Thus a hierarchically organized team of programmers tends to produce a hierarchy of programs (one main program or procedure which calls a number of subprograms or subprocedures) [Weinberg 71]. Today's systems are far more complex than the traditional file and transaction based batch systems; they are distributed, concurrent, and complex in their functions. We can not use the traditional hierarchical structure to produce new systems. This was expressed by Conway [Conway 68]: "...organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." Since the organizational structure determines the communication channels, the software systems development process must provide a communication network, which is flat (not hierarchical) and flexible. Therefore, project participants must be viewed as role players (or actors) for the particular task, who can change their roles from task to task, providing that they have the requisite qualifications.

Our communication model describes a network of communicating participants, who can change roles from task to task, and who have rapid access to project information. This open communication system changes the organizational structure of a development team to a flat and flexible network.

1.6.1 Communication Principle

We claim that communication is the most important aspect of the software systems development and that the quality of the software system depends on the effectiveness of the underlying communication system. Therefore, we have formulated the following principle:

> The *quality* of a software product depends upon the quality (effectiveness) of the *communication* system underlying the project development.

1.7 Outline of this Thesis

This thesis is organized as follows, Chapter 1 discusses the importance of communication in the software systems development process. It describes different types of communication processes and their roles in the systems development.

Chapter 2 provides an overview of the methods and models used in software engineering and systems analysis and design. It describes the SDLC models: waterfall, spiral, transformational, and object-oriented. Section 4 discusses the software process and software process programming. Section 5 introduces the *communication model*. The last two sections define the software systems development process and provide eight steps to the systematic modelling of the development process. They also model the software systems development process following Step 1 and 2.

Chapter 3 follows the remaining six modelling steps.

Chapter 4 is a refinement of Step 5 and it describes in detail a rigorous approach to the communication model. It specifies the three primary components: *participants*, *processes*, and *objects*. Objects are divided into *modules*, *knowledge repositories*, and *resources*. Section 7 of Chapter 4 specifies the waterfall, spiral, and evolutionary prototyping models using the communication paradigm. Thus, it demonstrates that the communication model is a high level and universal specification method.

Chapters 5 and 6 describe a case study in terms of structured and unstructured activities. Chapter 5 defines the task specification, whereas Chapter 6 provides more detailed specification for subtasks.

Chapter 7 discusses the general issues of formal specification: different levels and domains, abstraction, complexity, and multiple functions. This chapter serves as an introduction to Chapter 8, which describes the formalism of *Communicating*

Abstract Machines. Chapter 8 is a refinement of Step 6 – the formalization of the model.

Chapter 9 presents the conclusions: the contributions of our communication model to software engineering and future work.

Appendix A.1 includes the implementation details for the specification of a participant and a resource assignment procedure. Appendix A.2 describes specific services and mechanisms which can be implemented as a part of the Software Engineering Environments.

Chapter 2

Models for the Software Systems Development Process

> The problem of making computers useful to people as communications and information devices is not an engineering problem, it's a design problem. (M. Kapor)

2.1 Software Engineering and Systems Analysis and Design

Software engineering and *systems analysis and design* are relatively young disciplines and their subjects are still not well defined. Furthermore, some authors use the terms *programming-in-the-small*, *programming-in-the-large*, and *programming-in-many* referring to three levels of software: a single unit level (program), a system level, and an organizational level. However, there is no one-to-one mapping between these three types of *programming* and the two more traditional disciplines: *software engineering* and *systems analysis and design*. Defining such a mapping is difficult because software systems and their development have not only technical aspects, but also managerial, economic, social, psychological, and environmental aspects. Thus, they can be defined from many perspectives. The definition of software engineering varies between authors. To illustrate that, we quote three definitions of *software engineering*. The first one was given by Macro in [Macro 90]:

> Software engineering is the establishment and use of sound engineering and good management practice, and the evolution of applicable tools and methods, and their use as appropriate, in order to obtain — within known but adequate resources limitations — software that is of high quality in an explicitly defined sense.

Humphrey in [Humphrey 89] gives the following definition:

> Software engineering refers to the disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software.

Both definitions view software development as a well-organized and scientifically based activity. However, they describe software development as a *production* system. Important as it is, the production of software by itself is not the goal of the software systems development process. Rather, the goal is to provide adequate services to users. This notion is implied by Boehm's definition [Boehm 81]:

> *Software engineering* is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation.

This definition contains two key points. First, the subject of software engineering is not only computer programs, but also the related documentation, operational procedures, and manuals. Second, the created software system must be "useful to man." Therefore, the social, psychological, economic, and even political implications of the software systems are a part of the software engineering.

Systems analysis and design has a longer tradition than software engineering and it is concerned with various systems, not only software systems. It includes the study of a system and its problems, the definition of its requirements, and the evaluation of alternative solutions. As a result, systems analysis and design concerns many activities that are not a part of software engineering, such as feasibility studies, acquisition of software and hardware, acceptance testing, user training, and data conversion.

On the other hand, these two disciplines are interrelated. There is a growing tendency to extend software engineering to include all aspects of the software development process. This extension leads *software engineering* into the domain of *systems analysis and design.* Consequently, our communication model unifies both approaches.

The following three sections describe the software development *methods*, *models*, and *techniques.* The distinction between them is important. A method describes a high level approach to the systems development process and can use many models and many techniques. Whereas, a technique is based on specific tools and can be used with many methods.

2.2 Methods used in Software Engineering and Systems Analysis and Design

Although many authors refer to methods as methodologies, for example Clark in [Clark 90], we use the term *methods* as defined by Jackson and Dowson ([Jackson 87], [Dowson 86]).

Three methods prevail in *systems analysis and design* and *software engineering*:

- *Functional Decomposition*
- *Jackson System Design (JSD)* [Jackson 87]
- *Object-Oriented Decomposition* [Henderson-Sellers and Edwards 90].

All three are product-oriented; their progress is measured by the delivery of appropriate documents. The differences between these methods are based on different approaches to the system specification: (1) systems can be described primarily by processes (Functional Decomposition) or objects (JSD, Object-Oriented); (2) systems can be designed *top-down* (Functional Decomposition, Object-Oriented) or *bottom-up* (JSD).

The top-down analysis and design is built around a hierarchical structure of the system. In fact, few systems display such well structured organization. As Meyer [Meyer 88] points out: "Real systems have no top." However, Hatley and Pirbhai [Hatley and Pirbhai 97] claim that the systems in general exhibit hierarchical structure: "The models create a hierarchical layering of system specification, consistent with the hierarchical nature of systems in general." In practice, the system developers use both methods top-down and bottom-up simultaneously.

Functional Decomposition

Functional decomposition is a top-down analysis and design method. The term *functional* means that the system is defined as a set of interrelated functions. The data structures are secondary in this method. Functional decomposition works well with systems that have a simple hierarchical structure, and which are implemented using traditional procedural languages. This method is widely accepted by the industry and it is the basis for Structured Analysis and Structured Design (SA/SD). However, functional decomposition does not accommodate evolutionary changes, nor the iterative nature of the development process. Neither does it support the concepts of prototyping and reusable modules. It is also not very suitable for the analysis and design of distributed, reactive, and time-dependent systems.

Functional Decomposition Revised

Some structured methods, for example the Yourdon Structured Method (YSM) [Bowles 90], were significantly changed in the 1980s to expand beyond the functional specification. YSM uses *event partitioning* and state transition diagrams to represent control and timing.

Jackson System Development (JSD) Method

JSD is a middle ground between the process-driven methods and object-oriented methods. In JSD, the development process starts from the modelling of the real system (in application domain). The system functions are described after this model is created. Thus, JSD is different from the functional decomposition method, which starts from the specification of system functions.

Whereas the JSD method has two parts: specification development and implementation development, the traditional functional decomposition has three stages: analysis, design, and implementation. In Jackson's method, the traditional design phase is a part of the implementation.

JSD does not describe all development activities. For example, it excludes feasibility studies, project selection, hardware and software acquisition, project management, system acceptance, and user training.

Object-Oriented Decomposition

In the Object-Oriented paradigm ([Chen and Henderson-Sellers 90] [Meyer 88]), a system is described as a collection of *object classes*, which encapsulate both the data structure and the permitted operations. Objects communicate by sending messages. The implementation details of the objects and their procedures are hidden, while only the offered services are visible. Objects operate, therefore, on the *client/server* model, in which the client sends a request to an other object (server). The server executes an operation (or recursively makes a request for service) and sends back the results to the client.

Object-oriented analysis and design focuses more on data abstraction than on the procedural character of the system. One of the assumptions of the object-oriented design is that objects are changing slower than system functions; therefore, the object based systems require fewer changes than the systems designed around functions.

The object-oriented analysis, design, and implementation is fully beneficial when all three stages use the object-oriented model and an object-oriented programming language. Mixing different methods, for example: functional decomposition analysis, object-oriented design, and an implementation in traditional procedural language (COBOL) leads to complicated mappings between different paradigms.

2.3 Models for the Software Systems Development Process

Early models of the Software Development Life Cycle focused on the software product and its evolution through the phases. The first model, the *waterfall model*, was created to organize project activities. However, its strictly linear structure was not sufficient to model the natural iteration in development process. Later, a risk-driven model, *spiral model*, and new methods: *transformational*, *object-oriented*, and *parallel* were developed. On the other hand, new technologies and new applications gave rise to *rapid prototyping*, *exploratory programming*, and *system assembly from reusable components*.

2.3.1 Waterfall Model

The first model of the Software Development Life Cycle was strictly linear. The sequence of actions proposed by Royce in 1970 was widely accepted and became known as a *waterfall model*. However, it was soon discovered that the software development is incremental and iterative, and, as a result, the the traditional *waterfall* model was enhanced to include the iteration procedure.

The waterfall model consists of a sequence of phases. The number and names of them vary from author to author. Sommerville [Sommerville 89] includes five phases: requirements analysis and definition, system and software design, implementation and unit testing, system testing, and maintenance. Martin [Martin 90] introduces six stages: problem investigation, requirements analysis, feasibility analysis, design, construction, and changeover. In Martin's model maintenance (system support) does not exist. Whitten [Whitten et al 89] describes nine phases: survey, study, definition, selection, design, acquisition, construction, delivery, and system support. The system support phase corresponds to the traditional maintenance and improvement phase.

2.3.2 Spiral Model

The Spiral Model, introduced by Boehm in [Boehm 88], uses a *risk-driven* approach. It is based on the concept that the same sequence of steps is repeated many times throughout the system life cycle. Boehm describes five steps: (1) determine the objectives of the current cycle; (2) specify the alternative means of implementing the solutions; (3) evaluate alternatives, identify and resolve risks (for example, budget and schedule); (4) develop a solution and verify it; (5) prepare plans for the next cycle.

The main advantage of the spiral model is that it can be utilized by other models by adding the sequence of five steps to each phase. The risk-driven model can be used throughout the entire system life cycle; however, its results rely on the experience of the project developers to evaluate the risk factors.

2.3.3 Criticism of the Software Life Cycle

A number of articles have criticized the linear nature of the waterfall model and the concept of the software life cycle. The titles of some are self-explanatory, for example "Stop the Life-Cycle, I Want to Get off" [Gladden 82] or "Life Cycle Concept Considered Harmful" [McCracken and Jackson 82].

The criticism focused on three issues:

1. Life-Cycle phases are not applicable to all software projects.
2. Life-Cycle ignores the most important part of the development: communication between end-users and systems analysts.
3. All systems evolve, therefore we can not "freeze" the requirements. Furthermore, for most systems, we can not divide the system life cycle into initial development and a post-implementation phase, maintenance.

As a solution to this problems authors suggest the rapid-prototyping approach, which provides a working model of the system early in the process. Also, by using prototypes users can visualize and "play" with the system, which helps to formulate the system requirements.

2.3.4 Transformational Method

The transformational method is also called the *formal transformations* approach [Sommerville 89], *transform model* [Boehm 88] and *automated software synthesis* [Davis et al. 88]. It involves developing a formal system specification and then automatically transforming this specification into an operational code.

The transformational model allows for easier modification of the system since only the formal specification has to be modified. However, this approach is difficult to implement for large systems composed of subsystems using different paradigms and different programming languages.

2.3.5 Object-Oriented Method

A *fountain model*, described by Henderson-Sellers and Edwards in [Henderson-Sellers and Edwards 90], has six steps:

1. Identify the objects, their attributes, and the services they provide.

2. Establish interaction between objects (services required and services rendered).
3. Design detailed descriptions of objects.
4. Construct the library of objects.
5. Determine inheritance between classes of objects.
6. Construct the classes by aggregation and generalization.

The fountain model reflects the overlapping between steps and the iterative nature of the development process. However, it concentrates on the description of the project domain (application); not on the actual development processes.

2.3.6 Parallel Software Development Method

Conventional models (waterfall, spiral, fountain) have a single thread of activities, but some classes of systems, specifically high-risk systems, are developed in parallel by many teams. Therefore, the following *parallel* methods were introduced: *Dual development* [Ramamoorthy 81], *N-version programming* [Avizienis 85], and *N-fold inspection* [Martin and Tsai 90].

These methods allow for reduplication of one or more parts of the software development process. The *dual development* approach uses two teams working independently through each phase and consulting after each phase. In *N-version programming*, the requirements phase and the specification phase is done by one team, but the design and coding phase is done by N teams working in parallel. *N-fold inspection* uses N teams for the specification of users' requirements (revision of the requirements), while all the other phases are carried out by one team only.

2.4 Software Development Techniques

2.4.1 Rapid Prototyping

The *rapid prototyping* approach is based on a technique and a supporting environment which allows for building of rough models of the system. Prototyping facilitates understanding and the communication between users and developers and provides an iterative method of systems development. This approach includes several classes, for example, Davis [Davis et al. 88] describes the following three:

1. *Rapid throwaway prototyping*, in which a tentative and partial implementation is developed during the requirements specification phase. This approach is mainly used to clarify users' needs and facilitate communication between systems analysts and users.

2. *Incremental development* is a process in which the final implementation is constructed in phases and where a new functionality is added in each step. Incremental development assumes that most of the requirements are known at the beginning of the process, and implementation is performed in phases.
3. *Evolutionary prototyping* creates an implementation of known requirements and gradually adds new requirements, which are often discovered while using the initial prototype. In evolutionary prototyping, the requirements evolve in parallel with the prototype.

Since all projects use some form of prototyping, we view rapid prototyping as a technique rather than as a method. In practice, large projects are developed using traditional phasing approach mixed with the prototyping techniques. For example, Burns and Dennis [Berns and Dennis 85] describe a method combining waterfall model and rapid prototyping.

Exploratory Programming Technique

Exploratory programming is to some extent similar to *evolutionary prototyping*, in which requirements are specified along with the development of prototypes. Evolutionary prototyping creates the specification, whereas in the exploratory programming the implemented system itself is the specification. This technique is used in a specific class of systems, for example, AI systems [Sommerville 89], in which requirements are difficult to specify.

System assembly from reusable components

The *reusable* approach attempts to build a new system from the existing modules with minimal changes. The major emphasis is on production, description, and storage of the reusable modules. The reused modules are not only the code, but also specification and design modules [Goldberg 90].

2.5 A Process Description of the Software Systems Development Process

The traditional methods, models, and techniques capture only some aspects of the software systems development process. They describe the state of the documents (in the document-driven approach), the state of the code (in the code-driven approach), or the risk involved in the code production (in the risk-driven approach). However, they fail to describe the process itself. The need to define software systems development as a process was noted in the early 1980's. The First Workshop on Software Process in 1984 introduced the term *software process* [Balzer and Cheatham 84].

2.5.1 Software Process

There is no agreement between researchers on what should be included in the *software process.* Opinion varies from the very specific programming-in-the-small approach, through the broader programming-in-the-large (system approach), to the programming-in-the-many, which encompasses the managerial aspects of systems development. Moreover, five different terms are used to describe software systems development process: (1) *software process* (the annual international workshops), (2) *Software Engineering Process* [Humphrey 89], (3) *Systems Development Process* [Bostrom 89], (4) *Systems Engineering Process* [Humphrey 89], and (5) *Information Systems Development* (ISD) [Hirschheim and Klein 89]. The definitions of these terms are difficult to establish and there is no agreement as to which term should be used in which context. Therefore, we have decided to use a self-explanatory term: *software systems development process.* This process includes all development activities throughout the entire system existence.

2.5.2 Software Process Programming

Osterweil [Osterweil 87] stated in his seminal article that “Software Processes are Software Too” and consequently introduced the concept of *process programming.* Osterweil claimed that software development can be described algorithmically. His article started a lengthly discussion regarding process programming, its subject, models, and possible implementations. The various opinions can be divided into three groups: (1) process programming is realistic and its main problem is to find an adequate programming language [Katayama 89] [Kellner 89]; (2) process programming requires more modelling and empirical studies before it can be automated ([Notkin 89], [Curtis at al. 87]); (3) process programming is impossible, since it requires too much precision and it attempts to describe a non-deterministic process ([Lehman 87], [Lehman 89]).

For the last four years, researchers have tried many different approaches, models, and languages. The following list is not exhaustive; however, it demonstrates the diversity of frameworks:

1. *Functional* approach based on LISP [MacLean 89]
2. *Algebraic* approach [Katayama 89] [Nakagawa and Futatsugi 90]

3. *Logic* based approach, based on extended Prolog [Ohiki and Ohimizu 89] and fuzzy logic [Levary and Lin 91]

4. Process model based on *states* and *events* concept: *hierarchical communicating sequential tasks* (CSP like description [Hoare 85]) [Ashok et al. 89], State Change Architecture (SCA) [Phillips 89], and statecharts (STATEMATE) [Kellner 89]

5. *Object Oriented* approach based on Process Modelling Language [Roberts 89]

6. *Knowledge* representation framework [Borgida et al. 87]

7. *Dialogic* framework based on dialogue logic [Finkelstein et al. 89], [Finkelstein and Fuks 89]

8. *Contractual* model, ISTAR system [Lehman 85] [Dixon 88]

9. *Behavioral* models: behavioral approach ([Curtis 89], [Curtis et al. 88]) and Software Process Model (SPM) [Williams 88].

The *functional* approach models software process as a set of operations and rules. Similarly, the *algebraic* approach decomposes the process into activities, which are characterized by their inputs and outputs. The *Hierarchical and Functional Software Process* (HFSP) model represents complex activities as a hierarchical structure and describes them using mathematical functions.

In the *logic* based approach, processes are composed of subprocesses and are defined as predicates.

The *states* and *events* based models differ in their implementation. The *hierarchical communicating sequential tasks* approach describes objects, tools, user roles, and activities. An activity has precondition, action, state variables, and a structure of subactivities. Activities are instantiated by their parents. The statecharts based [Kellner 89] approach is similar, yet closely related to the visual formalism of Harel's Statecharts [Harel 87] and their particular implementation (STATEMATE). The model described by Phillips [Phillips 89], the State Change Architecture, is based on classes of objects representing *finite state machines*.

The *object-oriented* approach [Roberts 89] describes five principle classes: role, interaction, action, entity, and assertion. It is based on an object oriented conceptual modelling and Process Modelling Language, PML.

The *knowledge representation* approach is based on the premise that software development is knowledge-intensive and the application, system, and implementation knowledge must be captured. The DAIDA project, described in [Borgida et al. 87], organizes knowledge description in three layers: a requirements specification, a design specification, and an implementation specification. Each layer uses a different data model.

The *Dialogue* framework builds a systematic model of dialogue between clients and developers. It is based on the speech act theory.

The *Contractual* approach is, in its basis, similar to the dialogue framework — the software process is described by a set of *contracts*. *Contract* is defined by Dixon [Dixon 88] as "a well defined package of work that can be performed independently by a *contractor* (eg a developer) for a *client* (eg a manager or perhaps another developer)." The ISTAR system [Lehman 85] exchanges information between pairs of activities. The contractual model is restricted to the relationship between only two activities and a hierarchical structure of the system.

Behavioral models for the software development process focus on the group dynamics and, therefore, on cognitive, social, and organizational aspects. They also model learning and communication processes. Curtis [Curtis et al. 88] [Curtis 89] presents the results from empirical studies and describes problem areas. The *Software Process Model*, described by Williams [Williams 88], defines software development as a set of communicating activities. An activity is defined by a precondition, an action, a postcondition, and a message. Complex activities can be decomposed into subactivities.

2.6 Communication Model – a new Paradigm

The history of software engineering and systems analysis and design, described in previous sections, shows three tendencies:

1. A shift from *product-oriented*, models (document-driven or code-driven) to *process-oriented* models (risk-driven model and software process modelling and programming)

2. An expansion of the software process scope — from exclusively technical to economic, managerial, cognitive, social, and behavioral.

3. An attempt to formally define the software systems development process.

These trends lead, in turn, to two conclusions:

1. The model for the software systems development process can not concentrate exclusively on the technical aspects. In addition, it must describe the actual activities performed by people and machines, not the prescribed ideal procedures.

2. The software systems development process can be algorithmically described and executed by machines and people. Thus, the development of a software system can be viewed as a special type of software system.

Our *communication* model draws from many earlier described ideas and models. Specifically, it is related to the *behavioral*, *dialogic*, and *contractual* models. Furthermore, its formalization uses concepts similar to the *states* and *events* based approach. However, our communication model is not simply an extension of the *behavioral*, *dialogic*, or contractual approach – it is a new paradigm.

The communication model has three components: *communication*, *people*, and *services*.

Communication describes uniformly all development activities, which are executed by (abstract) *participants*: end-users, managers, project sponsors, systems analysts, application programmers, technical support groups, product vendors, standardization committees, quality assurance groups, technical writers, user trainers, steering committees, etc. Development activities provide *services* to participants.

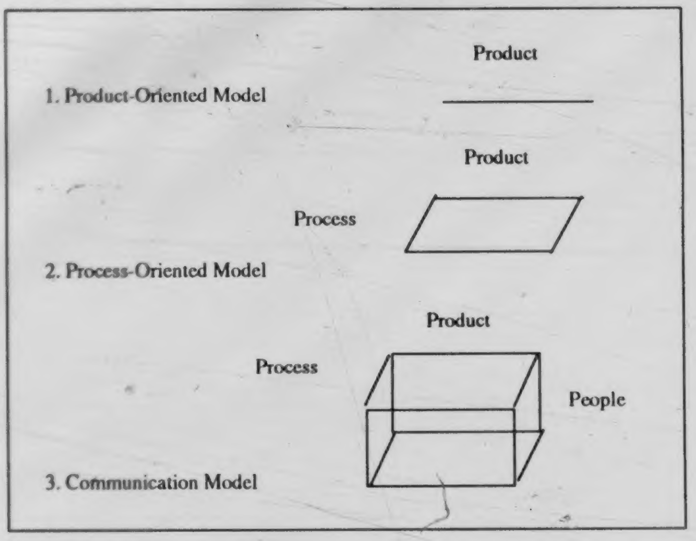

Figure 2: Evolution of Models for Software Systems Development Process

The evolution of the software systems development models is illustrated in Figure 2. The traditional models are centered around the *product*. They are either document-driven or code-driven approaches. The software process and process programming are process-driven; however, they concentrate almost exclusively on technical aspects. Although the *behavioral*, *dialogic*, and *contractual* models are concerned with the human aspects of the software development, they fail to provide an explicit representation for participants and human resources.

Comparisons with the Behavioral, Dialogic, and Contractual Models

The *behavioral* model was described by Curtis [Curtis 89] and Williams [Williams 88]. Curtis has presented the results from an empirical study of large software projects; however, he has not provided a model for the development process. He analyzed three problems occurring in projects: (1) lack of application knowledge, (2) changing and conflicting requirements, and (3) problems with communication and coordination.

On the other hand, Williams described a high level model, Software Process Model (SPM), which defines the development process as a set of activities communicating by sending messages. However, this model is lacking representation for the participants, resources, and products, and it does not describe management activities. Furthermore, the model does not include any formalization details. It does not express the *non-deterministic* character of the development process and does not support *iteration* of the processes [Armenise 89].

The *Dialogue* based model describes two parties (users and developers) involved in exchange of statements. This model is based on the speech act theory and it concerns exclusively oral communication. The dialogic approach focuses on the exchange of statements, not on the participants or the managerial issues (resources management, planning, scheduling, controlling).

The *contractual model* describes the tasks and specifies the resource requirements in terms of *efforts* – duration of a task in person-days. This model oversimplifies the human and managerial aspects of the development process.

The behavioral, dialogic, and contractual models capture some notion of organizational communication. However, they are not sufficient to describe the entire complexity of the communication in the development process. As well, they do not include an explicit representation of the project participants.

2.7 Communication – a Unifying Approach

The traditional models, methods, and techniques are centered around the product (the artifacts that are produced in the development process). As a result, they do not capture the most important part of the software systems development: communication. The communication model describes the real dynamics of the system development process. It offers a new perspective on systems development, while not excluding or replacing the existing methods and models. It gives a broader view on the entire process as well as allows for incorporation of other methods. Thus, it can be viewed as a high level uniform description for the existing methods (meta-model) or as a complementary model which can be used in conjunction with all other models.

2.7.1 Related Fields

The Communication model is based on a number of interrelated fields: software engineering, systems analysis and design, organizational communication (organizational behaviour), formal specification, office automation (groupware), database management, cognitive science (knowledge representation), and linguistics. The complex interdependences between these disciplines are illustrated by Figure 3.

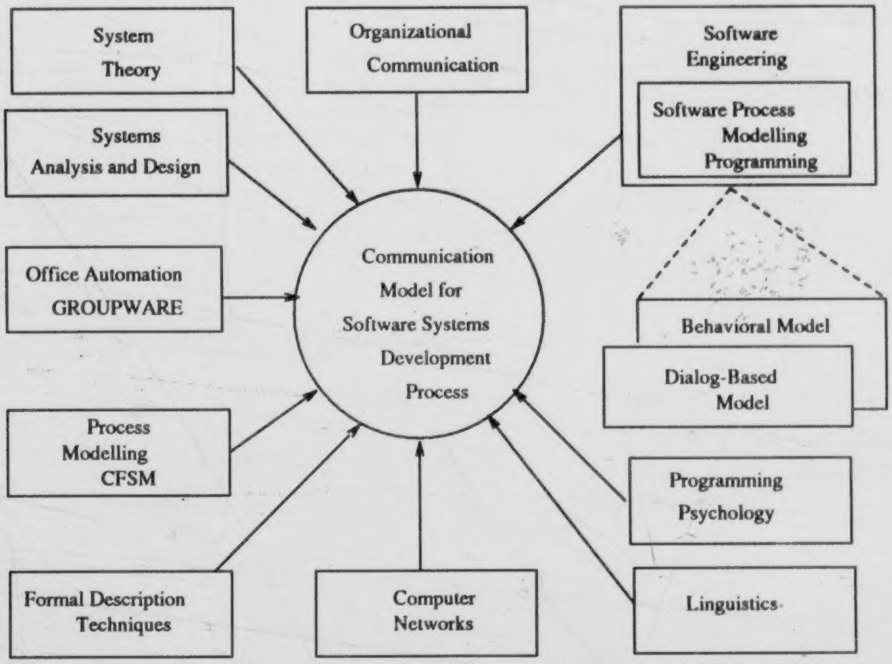
System Theory
Organizational Communication
Software Engineering
Software Process Modelling Programming
Systems Analysis and Design
Office Automation GROUPWARE
Communication Model for Software Systems Development Process
Behavioral Model
Dialog-Based Model
Process Modelling CFSM
Programming Psychology
Formal Description Techniques
Computer Networks
Linguistics

Figure 3: The Communication Model and other Fields

The next two sections describe (1) the relationship between the communication model and the *groupware* and (2) the relationship between the communication model and the *organizational communication.*

2.7.2 The Communication Model and the Groupware

Current software systems integrate two functions: *communication* and *information processing.* With a growing number of world-wide and organization wide networks, the software systems are shifting from primarily computational systems to communication systems [Cook et al. 91]. Computer networks and multi-media technology gave rise to the interdisciplinary studies of group dynamics and group communication: *Computer-Supported Cooperative Work (CSCW)*, and the software systems supporting group work, *groupware* ([Greenberg 91], [Ellis et al. 91]). Since both groupware and CSCW are emerging fields, their definitions have not yet been precisely stated. An approach to groupware varies from totally mechanistic (regarding groupware as a tool regulating and prescribing all communication activities) to totally unrestricted (regarding groupware as freely initiated interactions between participants) [Johnson-Lenz 91]. A vast number of different systems are called groupware — from shared file systems to *electronic classrooms.* Ellis, Gibbs and Rein give a definition and a functional classification of groupware systems [Ellis et al. 91]. They define groupware as: "computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment."

Their classification of the groupware systems includes six categories, to which we have added examples:

1. Message Systems (e-mail, multimedia e-mail, intelligent mail) [Gifford 90] [Borenstein 91]
2. Multiuser Editors [Ellis et al. 91]
3. Group Decision Support Systems (GDSS) and Electronic Meeting Systems (EMS) [Valacich et al. 91] [Grohowski et al. 90]
4. Computer Conferencing [Weedman 91]
5. Intelligent Agents [Gifford and Francomano 90]
6. Coordination Systems (form-oriented, procedure-oriented, and communication organization oriented models) [Olson and Bly 91]

The groupware systems are currently used on an experimental basis, mainly in academic and research centers. The subject literature describes a number of successful projects.

A Coordination system, Systems Concepts Laboratory (SCL) (developed by the Xerox Palo Alto Research Center (PARC)), has been experimentally used for three years in a distributed research environment. The analysis of this application was presented by Olson and Bly [Olson and Bly 91]. This project supported communication between two sites, one in Palo Alto, California and the other one in Portland, Oregon. Communication was based on computer technology and an open channel for interactive video and audio.

A computer mediated conference system was used at the University of California at Berkeley for communication between graduate students. The results were described by Weedman in [Weedman 91].

An example of the Electronic Meeting System (EMS), GroupSystems, was developed at the University of Arizona and described by Valacich, Dennis, and Nunamaker [Valacich et al. 91]. This EMS system has been in use for the last four years and has been tested in both laboratory and field settings in over 150 organizations (for example: Hughes Aircraft department and IBM's centers). The successful implementation of the EMS in over 33 of IBM's sites was reported by Grohowski [Grohowski 90].

A Relationship between a Communication Model and the Groupware

Our *communication* model should not be classified as a groupware. Although, both groupware and the communication model use the *communication paradigm*, they have different scopes and applications. In our work, we describe the software systems development process as a communication system. Therefore, we have introduced a rigorous communication model. In the construction of the communication model, the modelling process is of paramount importance; whereas the particular implementation of the model is a secondary issue.

However, there are two aspects that interrelate our communication model with CSCW and groupware systems. The results of the interdisciplinary studies of CSCW, specifically the studies oriented toward software projects, can be used to extend the communication model and build a set of specific communication protocols for particular classes of projects. Furthermore, the successful implementation of the groupware systems demonstrates that the *communication* paradigm is operationally, technically, and economicly feasible.

2.7.3 The Communication Model and Organizational Communication

Whereas the researchers and practitioners concentrate on technical aspects of the software systems development process, the behavioral aspects are relatively neglected. The majority of papers on the communication, negotiation, and learning aspects come from the management sciences ([Martin and Fuerst 84], [Bostrom 84], [Guinan and Bostrom 86], [Bostrom 89], [Koubek et al. 89], [Lind 87], [Montazemi 88], [Salaway 87]). The dialogue studies were concentrated mainly on human-computer interfaces [Harston and Hix 90], however, some works adapted the dialogue approach to software systems development modelling [Finkelstein and Fuks 89]. Other studies of group dynamics and group communication were done in conjunction with CSCW [Bostrom 88]. There are also studies from the linguistics perspective [Lyytinen 85].

The communication framework for the development of software systems was suggested over twenty years ago by Ackoff. Since then numerous works have suggested that effective communication is the most significant aspect of the system development. These works were concentrated around three approaches:

1. Classical information and communication theory (based on Shannon's theory) [Martin and Fuerst 84]
2. Empirical studies of communication in the project settings [Montazemi 88], [Bostrom 89]
3. Theoretical studies (modelling and analysis)

The models based on the communication theory have five components: sender, encoder, channel, decoder, and receiver. Participants send four types of messages: data, information, ideas, and decisions. Martin and Fuerst [Martin and Fuerst 84] analyzed the design process in terms of accuracy (transmission errors), timeliness (transmission time), efficiency (balance between input and output), and relevancy (relevancy to information, decision, etc.)

Bostrom [Bostrom 89] and Guinan [Guinan and Bostrom 86] stated that the software development process can be greatly improved by studying the existing communication patterns in the software development projects and by teaching the effective communication protocols.

The most difficult development activity is analysis of users' requirements, which are specified during an interaction between users and developers. Since these two groups operate in different domains, use different languages, and often represent different organizational culture, the communication between them encounters many difficulties. As a result, communication breakdowns are reflected in inadequate requirements. To improve communication and describe its patterns, Bostrom has used PRECISION model, which includes specific procedures (protocols) and behaviours for effective communication in the business setting.

Lind [Lind 87] described a model of organizational communication based on the Open Systems Interconnection model used in the computer networks. Each of the seven layers corresponds to an appropriate organizational level. For example, the highest level (application layer) corresponds to the users' environment and the lowest (physical layer) to the physical message sending.

2.8 A Systematic Approach to Modelling

Modelling of the systems development process operates in three domains: (1) the software systems development *process* itself – the problem domain, (2) the *model* of the development process – the solution domain (system domain), (3) the *implementation* of the model in a specific environment using specific language.

In general, researchers do not make a clear distinction between the *reality being modelled*, the conceptual *model of the reality*, and the *implementation* of the model. For example, Levary and Lin [Levary and Lin 88] describe the development process in both *process* and *model* terms and Liu and Horowitz [Liu and Horowitz 89] describe the problem space using concepts specific to the model itself.

It is very difficult to distinguish between the abstract description of the development process itself and its, equally abstract, model. However, the goal of the modelling is to separate the *modelled reality* from the particular modelling domain. The appropriate method of modelling has, therefore, three phases: (1) *analysis* of the problem domain, (2) *mapping* the problem domain into the model (solution), and (3) *mapping* the model into a particular physical implementation (artifact).

Therefore, we cannot formally specify process programming until we understand the software systems development process itself. Thus, we must first consider the *analysis* of the software systems development process, then proceed to *modelling*, and, following that, to a particular *implementation*. However, an opposite approach is taken in the current research on process modelling and process programming. With few exceptions, process programming concentrates on two areas: constructing a model (without prior analysis) or building an implementation (without a model or analysis). Therefore, the existing techniques lack a systematic approach; they do not include all necessary phases: *process analysis*, *model design*, and *model implementation*.

Analysis must abstract and describe the most important attributes of the development process, as well as, the basic types of activities, participants, objects, and the relationships between them.

A *model* should be built by mapping the characteristics, processes, objects, and relationships from the *process* domain into the *model* domain.

Taking into consideration the above guidelines, we define eight steps to a systematic modelling of the software systems development process:

1. *Define the overall scope of the modelling.* What general aspects of the process do we want to describe, analyze, model, and implement?

2. *Give the rationale for our work.*

3. *Refine the scope of the modelling.* Do we want to model the entire process or specific aspects? Do we want to have one high level model or many specific models?

4. *Describe the process and its model*

    (a) *Define the attributes* of the development process

    (b) Map the process attributes (Problem Space) to the model attributes (Solution Space).

5. *Build a rigorous model* and *verify it against the requirements* from Step 4.
6. *Formalize the model.*
7. *Describe examples* in the chosen formalism.
8. *Validate the model in the real world.* Implement the model and a chosen formalism in a large and complex project.

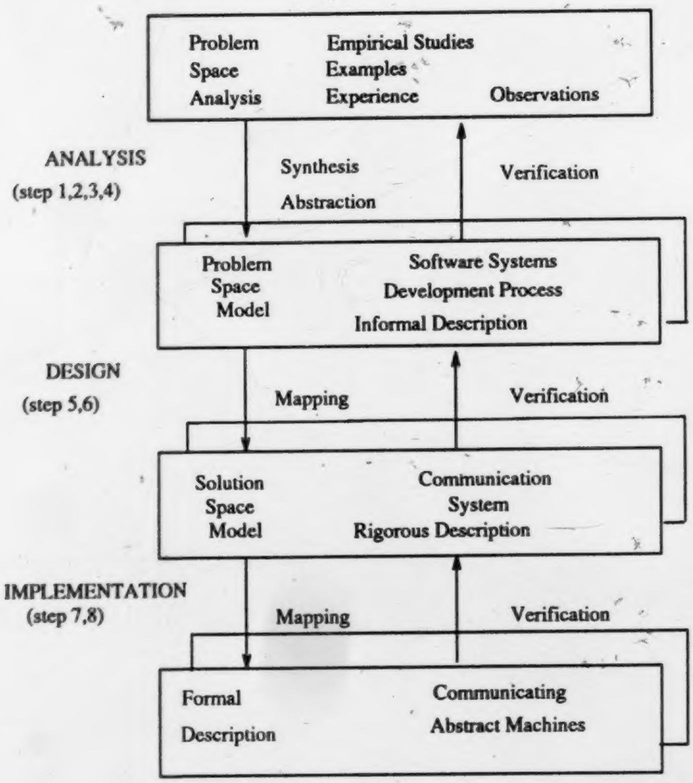
Problem Space Analysis
Empirical Studies
Examples
Experience
Observations
ANALYSIS
(step 1,2,3,4)
Synthesis
Abstraction
Verification
Problem Space Model
Software Systems Development Process
Informal Description
DESIGN
(step 5,6)
Mapping
Verification
Solution Space Model
Communication System
Rigorous Description
IMPLEMENTATION
(step 7,8)
Mapping
Verification
Formal Description
Communicating Abstract Machines

Figure 4: Modelling of the Software Systems Development Process

Figure 4 depicts the activities involved in the modelling of the software systems development process. These eight steps should be viewed as a way of organizing and describing the modelling effort, not as prescriptive normative rules.

2.8.1 Step 1 - Software Systems Development Process Definition

The software systems development process is defined here as a process describing the required services and providing tools to support these services. The software tools can be acquired or designed and built by the developers. Our definition emphasizes services not products:

> The *software systems development process* is a dynamic communication process which has three components: a *specification* of the required services, a *construction* of the products providing these services, and a *verification* of the constructed tools against their specifications. Specification, construction, and verification are repeated throughout the entire existence of a system.

We understand the term *development* as a process which continues throughout the entire life of a system. The changes are an inherent part of software systems and each change starts a new cycle of specification, construction, and verification. A software system should be easy to modify; this is the reason why *software* has been called *soft*. Balzer [Balzer 86] states that "Rather than attempting to eliminate the need for maintenance we should recognize that *enhancements*, not the initial development, is the *central software activity.*"

2.8.2 Step 2 - Why the Software Systems Development Process should be modelled?

In Chapter 1, we argued that without an understanding of the development process, large projects are developed using a mixture of some tested, yet not adequate, methods and intuitions. The results are often disastrous.

Modelling of the software systems development process provides the following: (1) understanding of the development process, (2) comparison methods for different development models, (3) methods and tools for knowledge representation, (4) quantifiable measurements for quality assurance, (5) support for project management, and (6) tools for learning about software development (for example: complete case studies).

Furthermore, the model implementation supports: (1) recording (documenting) project activities, (2) verification of the specified activities, (3) advising new participants about existing procedures, (4) automated execution of specific tasks, and (5) analysis of current status and project history (for example: status report or resource utilization report).

Chapter 3

Software Systems Development Process Modelling

3.1 Step 3 – Software Systems Development Process Redefined

The communication model has the following characteristics:

1. It represents all aspects of the development process: technical, managerial, legal, economical, political, social, psychological, and environmental.
2. It describes uniformly the entire life of the system; it does not differentiate between the traditional pre- and post-implementation phases.
3. It represents all development participants: end-users, managers, sponsors, systems analysts, programmers, technical support, technical writers, legal advisors, software and hardware product vendors, and librarians.
4. It defines software as *services* to people.
5. It is a generic high level model which is expressive enough to describe development activities in diverse projects. The need for such a meta-model has been noted, for example, by Deiters, Gruhn, and Shafer in [Deiters et al. 89].

3.2 Step 4 – Software Systems Development Process and its Model

The literature on process modelling and process programming presents many discussions regarding the definition of the development process. In this section, we describe four contributions ([Curtis 89], [Curtis et al. 1988], [Tully 89], [Liu and Horowitz 89], [Armenise 89]). Next, we integrate their perspectives and give a cohesive description of the software systems development process.

Curtis, Krasner, and Iscoe ([Curtis 89], [Curtis et al. 1988]) give the following list of problems occurring in the development process:

1. Lack of application knowledge among the developers. The application-specific knowledge must be captured and the project members must acquire this knowledge.
2. Fluctuating and contradicting requirements. The model must support changes (representation for uncertain design decisions, prototyping, change management, and control over change propagation).
3. Communication and Coordination problems. The software development environment must be used as a medium for communication to integrate people, tools, and information.

Tully [Tully 89] gives an alphabetical list of objects and concepts that should be modelled: actions, activities, agendas, agents, configurations, deliverables, events, messages, methods, obligations, permissions, pre- and post-conditions, roles, rules, tools, triggers, types, versions, and views. However, some of these items overlap the actual model, for example: pre- and post-conditions or agents. These two concepts belong to the *model* description, rather than to the development process description.

Liu and Horowitz [Liu and Horowitz 89] specify the following features for the process model:

1. Model must describe software development as a *design process*.
2. Model should be able to express parallel development processes.
3. Model should have a representation for the products (artifacts).
4. The activities should be executed only when the specified conditions are met.
5. The failed activities should be tractable.
6. A relationship between activities and resources should be explicitly stated.

Armenise [Armenise 89] describes the following characteristics of the software systems development process model: (1) it should be general, formal, executable, and open, (2) it should manage uncertainty and iteration; (3) it should record the rationale behind all project decisions; (4) it should support abstraction and parallelism; and (5) it should control the execution of activities.

3.2.1 The Characteristics of the Software Systems Development Process

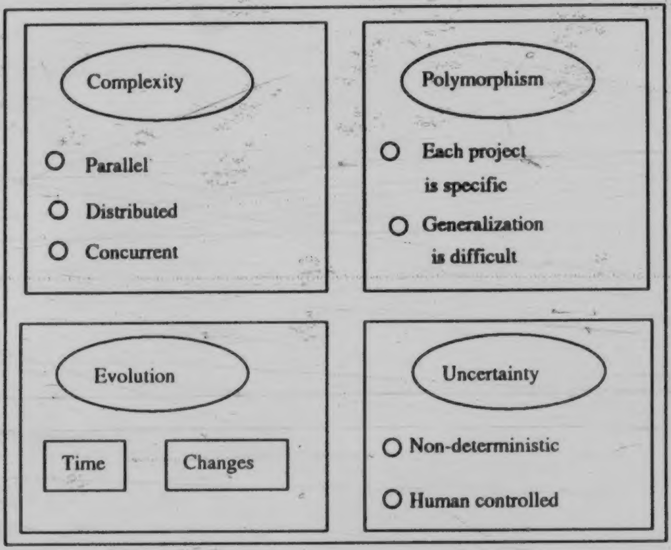
Complexity
○ Parallel
○ Distributed
○ Concurrent
Polymorphism
○ Each project is specific
○ Generalization is difficult
Evolution
Time
Changes
Uncertainty
○ Non-deterministic
○ Human controlled

Figure 5: Characteristics of the Software Systems Development Process

Figure 5 depicts, in our opinion, the most important characteristics of the software systems. In general, the *software systems development process* displays the following qualities:

1. *Complexity* — resulting from a large number of highly interrelated objects and processes. The complexity of a system is also related to the fact that the systems are concurrent, distributed, and real-time.

2. *Polymorphism* — each project is specific; therefore, it is difficult to define a generic development process.

3. *Evolution* — all systems are evolving; the changes may differ in their speed, but they are an intrinsic part of the system.

4. *Uncertainty* — development processes are executed by people, who react to external stimuli (for example: political or economic) and, sometimes, behave in totally unpredictable ways.

Complexity

The complexity of the software systems development process stems from two areas: (1) complexity of the software system and (2) complexity of the the process. The complexity of software systems is caused by a number of factors, among them: (1) the size of a software system, (2) complexity of the processes which are modelled by the software system, and (3) special characteristics of a software system, such as *parallelism*, *distribution*, or *concurrency*.

The complexity of the development process is caused by the following factors: (1) a large number of project participants and the high level of staff turnover, (2) negotiation problems (lack of user agreement regarding requirements), (3) distribution of the participants, (4) concurrency problems, and (5) changes in project policies and standards.

Polymorphism

Each project, as well as each software system, is unique. This specific quality does not allow for creating a universal method applicable to all projects for all software systems. However, a high level generic model can be used by different projects, which can extend this high level model by project-specific methods, techniques, and rules. On the other hand, some classes of development activities display many similarities. Therefore, their specifications can be reused.

Evolution

Since software systems exist in changing environments, they naturally evolve. Consequently, the development process changes together with systems. In addition, the development process evolves independently from software, by using new methods and technologies. In turn, the modifications of the development process often cause changes in the software systems.

The laws of *system evolution* were proposed by Belady and Lehman [Belady and Lehman 76]. These laws, often called Lehman's laws, are useful for describing software system dynamics; however, since their proposal, they have yet to be verified on sufficiently large systems for an extended period of time [Sommerville 84]. The *Law of the Continuing Change* [Belady and Lehman 76] states that "A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it." The *Law of an Invariant Work Rate* [Boehm 81] states that "The global activity rate in a large programming project is statistically invariant."

The second law seems to contradict the widely accepted fact that so-called maintenance activities constitute up to 75 % of a total system cost [CSTB report 90]. However, in most cases, the cost of maintenance does not include the enhancements or retrofits. Moreover, maintenance involves only the minimum necessary changes to source code, often excluding the corresponding changes to documentation, specification, and original requirements. In other words, the *system integrity* must be maintained. Otherwise, the complexity of the system increases with each change. This fact was expressed by Lehman's *Increasing Complexity Law* [Boehm 81]: "As a large program is continuously changed, its complexity increases, unless work is done to maintain it."

Uncertainty

Similarly to complexity, uncertainty has two sources: the software system and its development process. The specifications of software systems are deterministic only at specific points of time; otherwise, systems constantly evolve in unpredictable directions. This characteristic is reflected in the development process, which has to accommodate the incoming changes and, at the same time, maintain the integrity of the overall system. The uncertainty of the development process is unavoidable, since people make the decisions and control the system.

3.2.2 Components of the Development Process

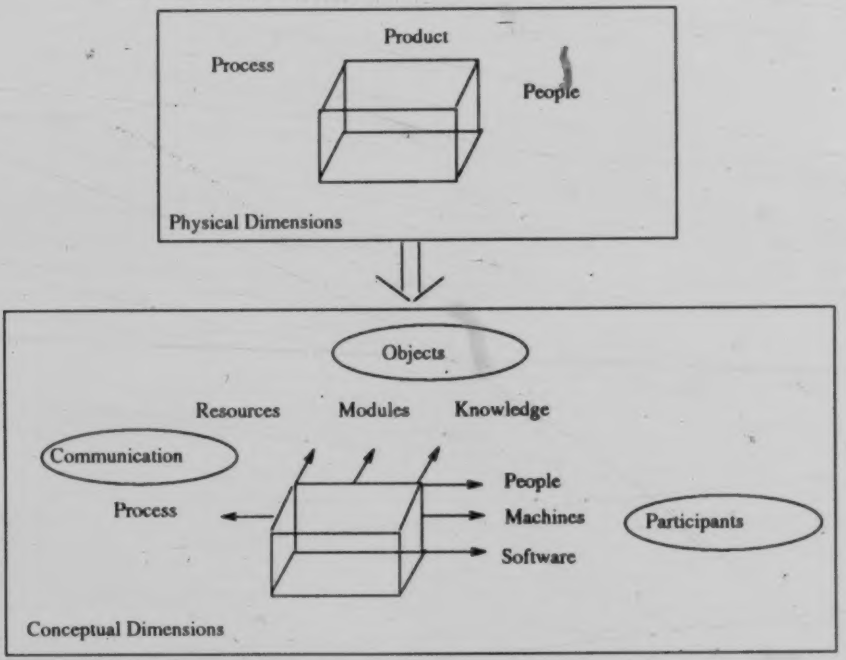
Product
Process
People
Physical Dimensions
Objects
Resources
Modules
Knowledge
Communication
Process
People
Machines
Software
Participants
Conceptual Dimensions

Figure 6: Mapping between the Development Process and its Model

Figure 6 presents the mapping between the *people*, *products*, and *processes* and their abstract counterparts: *participants*, *objects*, and *communication processes.*

Four types of processes are present in all projects: (1) *communication*, (2) *negotiation*, (3) *learning*, and (4) *managing*. The *communication* process is a generic activity, which underlies negotiation, learning, and managing.

In our model, participants specify, modify, execute, and control processes. They are described in terms of their *roles*, not in terms of actual physical people. Thus, participants roles can be performed by one person, a group of people, or a machine.

Objects can be divided into: *resources*, *knowledge*, and *modules* (services). Resources include people, time, finances, hardware, and software. Knowledge is a collection of information and rules required to perform development activities. Modules are the physical objects created or modified by the the process, for example: specification, source code, or test documentation.

The Problem Space and the Solution Space

The mapping from the development process attributes to the particular model requirements is not one-to-one. Often, a single characteristic is represented by a number of modelling concepts.

The *polymorphism* problem is solved by the *descriptive* nature of the communication model and *abstraction* levels. This generic model can be converted into specific or prescriptive models by adding a set of rules (constraints) for specific projects.

Complexity of the system is expressed by two notions: *abstraction* and *granularity* in the model.

Evolution of the process is supported in the model by the notions of *granularity*, *abstraction*, *timing*, and *traceability.*

Participants have a straightforward representation in our model. The *processes*: communication, negotiation, learning, and managing, are described by a generic *communication process*. *Objects* are mapped into corresponding objects.

3.3 Step 5 - A Rigorous Model for the Software Systems Development Process

We describe the *software systems development process* as a collection of participants communicating by sending messages. Interactions between participants are specified by the communication protocols. Participants are the only active component in our model, and they manipulate objects. Thus, objects are created, stored, modified, deleted, analyzed, or sent by participants.

The communication system, S, representing the software systems development process, is defined as a triple:

$$S = < P_s, O_s, C_s >$$

where
P_s is a finite set of participants,
O_s is a finite set of objects,
C_s is a finite set of communication processes.

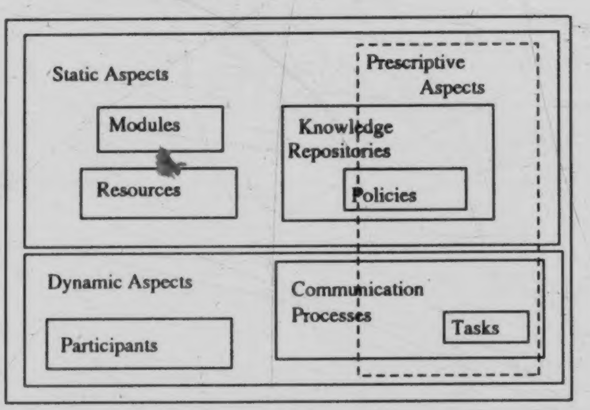

Figure 7: Three Aspects of the Communication Model

Figure 7 illustrates the relationships between the *dynamic*, *static*, and *prescriptive* aspects of the communication model. The *dynamic* component is represented by *participants* who are the initiators, executors, and controllers of processes. The *static* aspect corresponds to the objects sent between participants (only the *access rights* are sent) and stored in a common repository. The *prescriptive* component is represented by a set of policies. They are stored and used for creation and modification of processes. Thus, the processes reflect the project policies. Policies are shown in Figure 7 as a subset of the knowledge repositories. However, this particular placement should be viewed as an example of an implementation, not as a part of the conceptual model itself.

Granularity

Granularity determines the level of detail in an object or process description. An object or process can be defined at many levels by adding *horizontally* or *vertically* the necessary details.

Horizontal expansion translates into the addition of new participants and protocols to the same level of description. Horizontal decomposition breaks the participant, process or object into a number of orthogonal modules, which ideally should be loosely coupled.

Vertical expansion adds more detailed descriptions at the lower levels. It determines the composite and atomic elements and creates hierarchical structures of components.

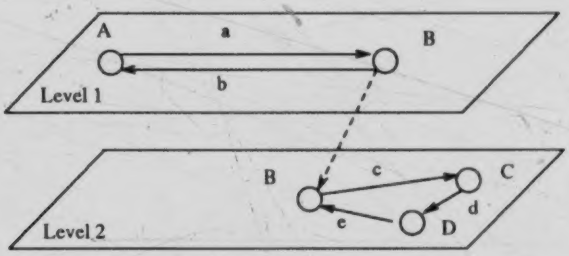

Figure 8: Vertical Expansion of the Process Specification

Figure 8 depicts the vertical decomposition of a process with two levels: *Level 1* and *Level 2*. In the graphical representation, the planes correspond to levels, the circles represent participants, and the arrows represent flows of messages. Level 1 describes the following situation. Participant A is working on an interface for an accounting system. When the design is finished, participant A sends all related documents for authorization (message a) to participant B. Participant B receives the request, performs appropriate procedures (authorization procedures are hidden at level 1) and sends back either authorized documents or rejections (message b). The Level 1 procedure is expanded vertically by adding Level 2, which describes in detail the authorization process itself. Participant B sends the request for authorization (message c) to the steering committee, which authorizes or rejects the documents (this process is hidden at level 2) and sends the results (message d) to participant D, a quality assurance group. D sends the verified authorization (message e) to B.

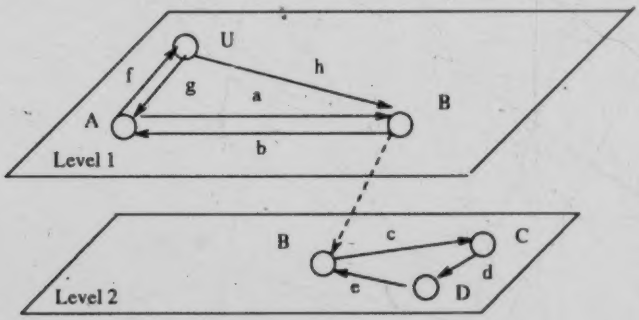

Figure 9: Horizontal Expansion of the Process Specification

The described process can also be expanded horizontally by adding new participants to Level 1 (Figure 9). The description of this process has been changed in the following way. Two participants, the analyst A and the user U, work together on the system interface. The communication between both of them is represented by messages f and g. When the design is finished, A sends the documents for authorization (message a) and U sends an independent evaluation report (message h) to B.

Abstraction Levels

Abstraction distinguishes between the ***description*** (specification) of a class of participants, processes, or objects and a particular ***instantiation*** of a participant, a process, or an object.

Objects have their abstract specifications and actual occurrences. Thus, modules, which are a class of objects, have their specifications and their physical counterparts. For example, a module can have the following specification: French translation of Hoare's book Communicating Sequential Processes. The physical occurrence of this module is the actual French text (artifact).

Participants are abstract specifications, which describe particular *roles* played by people or machines. A role description is merely a specification for an actor. For example, a fluent French-English translator who specializes in computing science translations; a publisher; and a technical revisor are specifications for three participants. The physical occurrence of the first participant can be a specific person A, a group of translators, or a software system providing translation services.

A process (task), in our model, has two aspects: ***specification*** and ***execution*** of the specification.

Specification of the development processes involves two steps: (1) decomposition into communication subprocesses and (2) formal specification of the subprocesses. On principle, project activities should be formally (rigorously) specified and verified before they are executed. However, the extent of formal specification is project-specific. Processes are specified and modified by human participants; although, computer support can facilitate this process. In general, all changes to process specification must be controlled and recorded. The change documentation must include not only a change description, but also the ***rationale*** for the particular change. In many respects, task specification can be compared to source code of a computer program.

Execution of the process incorporates management activities: planning, scheduling, resource allocation, and controlling. Tasks are executed under ***human*** control; however, specific classes of tasks can be partially or even fully automated. Furthermore, each execution should be recorded, so that the processes are ***traceable***. To some degree, process execution can be compared to interactive or real-time program execution.

Traceability

Although the communication system is not a *document-driven* system, documents are a crucial part of our model. We have introduced two rules to govern documentation: (1) processes must be specified and verified before their execution and (2) executions must be traceable. These rules ensure that project documentation is created automatically, as a *by-product*, and in parallel with the actual activities, not, as still often happens, as an afterthought.

Timing

Time is used in the development process in many functions; for example: to control multiple versions, synchronize processes, or estimate task duration. Since the details of timing are implementation-specific, we discuss time related issues in Chapter 8, as a part of the formalism description.

Process Representation

All processes are specified by communication protocols, which are defined and modified by participants.

Processes can be specified *top-down* or *bottom-up.* The *layered architecture* is based on the *client/server* paradigm, in which higher level *client process* requests a *service* described at a lower level. Service details are hidden from the higher level description. This architecture, also called an *encapsulated specification*, provides a changeable and expandable model.

In the *top-down* specification, a process description starts with a high level, for example, a testing process. Next, the process is decomposed into subprocesses. The testing process can be decomposed into four subprocesses: test plan preparation, unit test, system test, and acceptance test (test performed by the user). Subsequently, subprocesses can be decomposed into lower level subprocesses. In our example, the first subprocess, test plan preparation, can be factored into three subprocesses: operational test preparation (critical services, exceptions handling), performance test preparation (response time), human engineering test preparation (human interface). This decomposition procedure can be repeated until the specification is satisfactory to the project participants.

In the bottom-up specification, the process description starts from the atomic processes which are then aggregated and abstracted to higher level complex processes.

Participant and Object Representations

The *participant* representation has two levels: (1) *conceptual* – participants' roles and (2) *physical* – instances of participants. Conceptual specification is a high level description of skills, experience, and knowledge, which are necessary to execute a particular service. The actual physical participants are the people, machines, or processes. A physical person can play different rôles and execute many tasks in parallel. For example, the same person can play a role of project leader in process A and a role of application programmer in process B.

Objects and their representation are described in Chapter 4.

3.4 Step 6 - Formal Specification

The communication model is based on the thesis that the model for the complex software development process must include an explicit specification of the formal communication between project participants. The communication processes should be *formally* (or rigorously) specified and the history or their executions should be well documented. It is not the intention of our model to introduce too much rigor; rather, the goal is to clearly specify and analyze the procedures already used by a project and then automatically support their executions.

Contrary to popular opinion among practitioners, formal methods are used not only in the academic or research environments; they are also used with very good results by commercial systems. For example, Hall [Hall 90] states the following facts, based on his experience with a large software engineering company:

1. Formal methods are helpful in finding and eliminating errors in early stages of analysis and design.
2. Formal methods provide rigor and organization to system specification.
3. Formal methods are based on mathematical concepts which are at a higher level of abstraction than the programs, and therefore are easier to understand.
4. Writing a formal specification decreases the cost of development.
5. Formal methods help the users understand the specification of the system.

The Degree of Task Formalization

Ideally, all development activities should be formally specified and verified. However, for most projects, this approach would be not feasible. Therefore, our communication model is designed as an *open* system, which can be incrementally built from atomic specifications. As a result, the extent of formalization depends on participants of particular projects.

A Formalism for the Communication Model

Since our model is based on the *communication* paradigm, it has many similarities with communication networks. Furthermore, the description of the communication process is based on a concept of *protocols.* Protocols are used here in a similar way to the computer network protocols. Therefore, in searching for an adequate formalism, we have investigated standardized *Formal Description Techniques* for computer networks protocols. The currently existing standards use three formal languages: Estelle ([NBS Report 87]), Lotos ([Turner 89]), and SDL. The International Standards Organization (ISO) standardized *Extended Finite State Machine Language*, Estelle, and *Language Of Temporal Ordering*, Lotos. The CCITT has standardized a *Specification and Description Language*, SDL. Whereas Estelle is based on the *communicating finite state machines* formalism; SDL is based on similar concepts of *extended finite state machines.* LOTOS specifications are based on temporal logic.

The state-based models are widely used in software engineering and are very popular in other disciplines. Therefore, we base our formalism on the Communicating Finite State Machines (CFSM) and Estelle. The implementation of a well known specification technique has major advantages. The literature and a number of applications have demonstrated that the FSM-based model can be successfully used for communication protocols. Furthermore, the existing Estelle's facilities: verifier, simulator, and graphical representation, can be adapted and used by the particular implementation of our model.

3.4.1 Organizational Advantages of the Communication Model

The communication model can improve the following aspects of project management:

1. Increase the accuracy of the users' requirements specification by applying precise communication techniques.

2. Eliminate unnecessary communication. Since general communication rules are explicitly stated and critical communication tasks are formally specified, tasks can be checked for unnecessary redundancy or discrepancy.

3. Improve the understanding of the tasks and the participant roles throughout the entire project. Participants can view the task specifications, the history

of task executions, and the current task status.

4. Reduce job related stress by clearly stating the responsibilities of participants.
5. Provide automated documentation of project task specifications and executions.
6. Assist new project participants in the learning process by providing access to previous task executions as well as to current task specifications.
7. Overcome the culturally and socially prescribed communication patterns. Grohowski, McGoff, Vogel, Martz, and Nunamaker [Grohowski et al. 90] report that the anonymity of electronic meeting systems helps to overcome the status and cultural differences interfering with productivity.
8. Facilitate communication between participants from remote sites. Formally specified tasks and clearly stated responsibilities help to organize work in terms of part-time jobs and work from the home office.

3.4.2 Formal Model and an Automated Task Verification

The formal specification of communication processes provides tools to identify some of the properties and to perform task analysis. The existing CFSM-based methods can be used for the following:

1. Reachability analysis
2. Deadlock detection
3. Completeness checking (for example: balancing the sent and received messages)
4. Identification of critical paths in the tasks scheduling process
5. Identification of communication *bottlenecks*.

3.5 Step 8 - Implementation of the Communication Model

The following section demonstrates that the implementation of the communication model is operationally, economically, and technically feasible. The proposed model is *modular* which means in practice that it can be built from simple blocks. The extent of task specification is totally dependent on the size of a project, its specific type, and the type of people involved in the project. *High-risk* systems, as well as large and complex systems require a more rigorous approach. Small projects can utilize formal protocols for only a single area, for example requirements specification.

The communication process is dynamic but it also reflects the organizational structure of a particular project. The model itself is seen as a box of tools to build customized software development environments. The users of the model can use the standard tasks or they can specify their own development tasks. Each of the project management activities can be provided by standard protocols or redefined to meet particular needs. The resource assignment procedure can easily be modified to incorporate project or company specific policies. In decision making tasks, company policies are imposed by the constraints. During the specification of tasks (before tasks execution) the constraints are verified. For example, let us assume that a project has rule $R1$ (constraint): "all tasks involving financial statements must be approved by the accountant (participant playing the role of accountant) before their execution." This rule will be implemented in the following way. During task t specification, the verification mechanism will identify the pertinent rules. Since task t involves a financial statement, rule $R1$ will be applied, and the task t specification will include an accountant approval subtask $t1$. The subtask $t1$ will be triggered during the task scheduling procedure. The Approval Request message will be sent to the accountant participant. Once the task t is approved (the approval answer is sent from the accountant), the task t will be scheduled for execution; otherwise the results will be sent to the task initiator (participant who has requested the task scheduling).

The communication model will be implemented as a computer based network of communicating agents. All objects, including paper documents and images, will be stored in databases and will be accessible by all participants (with some limitations). Messages will be sent according to protocols and distributed to participants, for example, by electronic mail.

Chapter 4

A Rigorous Model For The Software Systems Development Process

The *software systems development process* model has three primary components:

1. Participants
2. Processes
3. Objects: modules, knowledge repositories, and resources.

Participants are active agents who make decisions, control tasks, exchange messages, and perform services.

Processes describe development activities, which are executed by the participants upon objects: *modules*, *knowledge repositories*, and *resources*. Processes are created and modified according to the policies described in the *knowledge repositories* and executed utilizing the assigned *resources*.

Tasks are processes with *desired outcomes*. Thus, a *process* can be completed, but the *task* can fail. For example, in Chapter 3, we have described an authorization process, which has two possible results: "authorization" or "rejection." However, the authorization *task* requires one outcome only: "authorization." Hence the authorization process can be repeated successfully many times, yet, in each execution, the authorization task can fail.

Modules are the objects produced or modified in the development process. They include all project documents (for example: feasibility assessments, problem statements, requirements specifications, and requests for proposals), data dictionary, user manuals, and application software.

Knowledge repositories contain information about the application system, development methods and techniques as well as software and hardware.

Resources describe objects used in the development process: human and financial resources, time constraints, available hardware and software, equipment, etc.

The organization of this chapter is as follows. The first section defines a *project* and its ***specification***, ***execution***, ***modification***, and ***history***. The subsequent sections describe: ***participants***, ***tasks***, ***modules***, ***knowledge repositories***, and ***resources***. Section 4.7 uses the communication model to represent the waterfall model, spiral model, and the prototyping technique. Examples of participant specification and resource allocation procedures are included in Appendix A.1. Implementation of services and mechanisms is described in Appendix A.2.

4.1 A Project Definition

The communication model of the software system development process will be implemented by particular projects. Thus, we describe the model in a project setting. A software *project* is defined by two notions: *project specification* and *project execution*. The project specification defines sets of allowable participants, tasks, modules, knowledge repositories, and resources for a specific software development project. The project execution describes a particular instance of a project specification performed under well-defined conditions, within a specified time, by designated resources, and involving active tasks.

A project specification is defined as a tuple:

$$Project\ Specification = < P, T, M, K, R >$$

Where
P is a finite set of all project participants,
T is a finite set of all rigorously specified tasks,
M is a finite set of all project modules,
K is a finite set of all knowledge repositories,
R is a finite set of all allowable project resources.

A project execution is defined as a tuple:

$$Project\ Execution = < P_e, T_e, M_e, K_e, R_e, \mathcal{F} >$$

Where
P_e is a set of project participants involved in at least one currently active task, $P_e \subseteq P$,
T_e is a set of active project tasks, $T_e \subseteq T$,
M_e is a set of active project modules, $M_e \subseteq M$,
K_e is a set of active knowledge repositories, $K_e \subseteq K$,
R_e is a set of assigned resources, $R_e \subseteq R$,
$\mathcal{F}$ is a mapping assigning project resources to project participants, $\mathcal{F} : R_e \rightarrow P_e$.

A Modification of the Project Specification and Execution

A model for the systems development process must be *expandable* and *changeable*; therefore, the project specification and its execution must be modifiable. Project components are closely interrelated, a change to one component can propagate throughout the project specification and execution. Thus, modifications to processes, participants, or objects involve complex propagation techniques. Since these techniques are implementation specific, we do not give more details at this level and we conclude our discussion with an example of a change to human resources.

In general, software projects have a high staff turnover. Thus, the situation, in which an experienced project member resigns unexpectedly, is not unusual. In this case, the data related to a particular team member is removed from project resources and all tasks involving that person are reassigned or suspended. Subsequently, the reassignment of tasks often causes a *ripple effect*, which spreads the changes throughout the project. On the other hand, projects with sufficient time or financial resources avoid task reassignment either by postponing the completion dates for affected tasks or by hiring a replacement. This example does not exhaust all possible solutions to staff fluctuation; however, it indicates the complex interrelations between tasks and resources.

4.2 Participants

Our communication model gives a uniform representation to the various groups and individuals participating in the project development. It models the activities performed by the users, the customers, middle and executive management, steering committee, system developers, system support team, advisory committee, industrial standards enforcement group, and many other groups and organizations. In general, *participants* represent the activities executed by a person, a group, or a machine.

Project participants are the actors who perform four functions:

1. Send and receive messages.
2. Make decisions (most of the decisions are external to the system).
3. Control task execution (initialization, termination, and modification).
4. Perform requested services.

Participants do not describe physical persons; they define their functions in a communication process. This role-oriented approach has two advantages: organizational and psychological. Organizationally, the separation between roles and physical persons allows for an easy task reassignment. Psychologically, it facilitates the negotiation process between individuals with different backgrounds and different requirements. Depersonalized task description is a part of Boehm and Ross's *Win-Win Theory* [Boehm and Ross 1989]. Our approach follows the three rules of the *Win-Win Theory*: it separates the people from the problem; focuses on people functions not on their positions; and uses objective criteria for staffing.

4.2.1 Participant Definition

A *participant* is defined as a tuple:

$$Participant =< participant\ identifier, A >$$

where the *participant identifier* is a unique identifier and A is a finite set of attributes.

A special class of participants, *librarians*, is responsible for the three repositories: *knowledge*, *modules*, and *resources*. Librarians maintain the data according to project standards and provide information and modules upon requests.

4.3 Tasks

Project tasks specify development activities and their desired outcomes. Processes describe all predictable communications between task participants. However, in general, the final results of the tasks are non-deterministic. This non-determinism has two sources: (1) the task participants make decisions external to the system and (2) an external *control* process can modify or terminate task execution.

Functionally, tasks describe the creation of participants, modules, knowledge repositories, and resources. In particular, task participants execute a special *creation* task to specify other tasks. Task execution can be modified by the participants through the *control task*.

Structurally, tasks have a *layered architecture* in which a higher layer (client) uses the services provided by a lower layer (server). The *client/server* organization of tasks provides a hiding mechanism. Thus, the organization of lower levels is invisible to the higher levels and the services are supplied through an interface.

In terms of predictability, tasks can be divided into two classes: highly predictable tasks, called *structured tasks*; and unpredictable tasks with changing specification, called *unstructured tasks*.

The following subsections describe task *specification*, *execution*, *state*, and *history*. *Task specification* describes all possible communication activities of task participants. *Task execution* is a physical occurrence of task specification. The *task state* is defined by the current state of each participant. *Task history* records task states from the initial to current state.

4.3.1 Task Specification

A task description has two components: (1) *process* specification and (2) *desired outcome* specification.

$$Task\ Specification = < Process\ Specification,\ Desired\ Outcome\ Specification >$$

First, we present the definition of process specification. *Processes* involve sets of participants, communication protocols, project modules, and knowledge repositories. Communication protocols describe the behaviour of a process.

Process specification is defined by a tuple:

$$Process\ Specification = < P_t, T_t, \{Attribute\}_{i=1}^{n}, M_t, K_t >$$

Where
P_t is a set of participants involved in the specified process.
T_t is a set of communication protocols describing the behaviour of the participants,
$Attribute_i$ is an implementation specific process attribute, for example: *priority*.
M_t describes the set of project modules affected by the specified process.
K_t describes the set of knowledge repositories affected by the specified process.

Task specification is defined by a tuple:

$$Task\ Specification = < P_t, T_t, \{Attribute\}_{i=1}^{n}, M_t, K_t, O_t >$$

Where O_t is a finite set of desired outcomes, described in terms of the communication processes. Thus, the description of a desired outcome includes all details pertaining to an interaction, such as: sender, receiver, time, message content, and other message conditions.

Task execution is defined by a tuple:

$$Task\ Execution = < task - id, P_e, T_e, M_e, K_e, O_e, R_e, \mathcal{F} >$$

Where
task-id is a unique task identifier,
P_e is a set of participants involved in the task execution
T_e is a set of protocols describing the behaviour of participants
M_e is a set of project modules affected by the task execution
K_e is a set of knowledge repositories affected by the task execution
O_t is a finite set of desired outcomes
R_e is a set of the project resources required by the task execution
$\mathcal{F}$ is a mapping between project participants and project resources, $\mathcal{F} : P_e \rightarrow R_e$.

4.3.2 Task State

The global task state is described by the state of all task participants and communication channels.

Task state is defined as a tuple:

$$Task\ State = < < s_i >, C >$$

Where
s_i describes the state of the participant P_i
C describes the priority queue for the messages (mailbox) for all task participants.
Figure 10 shows the relationships between task specification, scheduled task, and task execution.

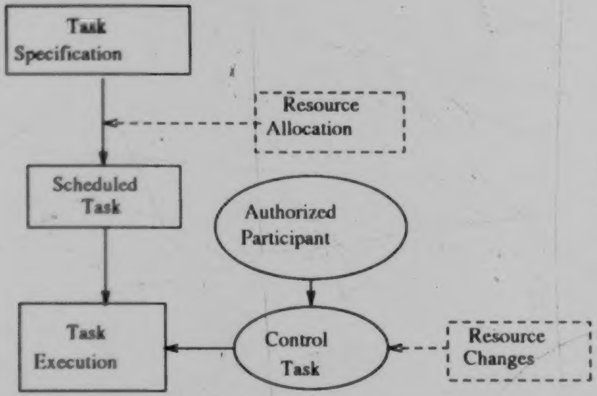
Task Specification
Resource Allocation
Scheduled Task
Authorized Participant
Task Execution
Control Task
Resource Changes

Figure 10: A Task Life Cycle

4.3.3 Task History

Task execution is described by a trace of the process. Thus, a *task history* is a sequence of task states.

In a particular implementation, a task execution can be recorded by the *history mechanism*, which registers a system time for each state. In this timing method, there are two recorded points: start time and end time.

The *task history* is defined as a finite set of triples :

$$Task\ History = \{< task\ state, \tau_{start}, \tau_{end} >\}_{i=1}^{n}$$

Where
task state is one of the valid task states
τ_{start} represents the state *start* time
τ_{end} represents the state *end* time.

4.3.4 Control Tasks

In general, tasks can be controlled from many levels. For brevity, we present an example with two control levels: *task* and *project*. A single task execution is monitored by the *Control Task* (Figure 10), which has an *authorized* participant who can initiate, suspend, restart, terminate, or abort task execution.

At the higher level, all tasks are controlled by the *Project Control Task*, which can interrupt other task executions, change their specifications, and restart their executions with new specifications.

4.4 Objects

Objects are *passive*; they can be created (modules) or they can be used by processes (knowledge repositories and resources).

4.4.1 Modules

Project modules are defined as self-contained permanent documents, such as: user requirements, system specifications, design architecture, user manuals, data dictionaries, stored data, and application programs. Modules are created and modified in the system development process. All documents are included in the same class: project modules. This approach recognizes two important aspects of system development. Firstly, with increasing usage of formal specification languages, the system specification can be automatically converted into computer programs. Secondly, system documentation is as important as programs.

A module has a unique identifier, date-time stamp, last task identifier, and module description. Modules are stored in a module repository and maintained by librarians. The librarians provide copies of modules upon authorized request. Thus, a participant must have specific *access rights* to obtain a copy of a module.

4.4.2 Knowledge Repositories

Knowledge repositories involve three classes of information: *application knowledge*, *software and hardware knowledge*, and *systems development methods and techniques*.

The *application knowledge* repository includes information about the system environment, users, organizational structures, finances, and application.

The *software and hardware knowledge* repository contains information about the software packages, programming languages, operating systems, hardware, and computer networks related to the project.

The *methods and techniques* repository describes models, specification languages, and verification methods used in the project.

4.4.3 Project Resources

Project resources include human, hardware, software, and financial resources. They also include special equipment and time constraints. A resource is defined by the following tuple:

$$Project\ Resource =< Resource\ Id,\ Resource\ Type, \{Attribute\}_{i=1}^{n} >$$

Human resources describe the people involved in the project development. Hardware resources describe the hardware available for a particular project. Software resources describe the acquired software, which is used by the project without change. If the source code is available and changed by the project, the software resource becomes a project module. Equipment resources include office space, desks, chairs, tables, overhead projectors, etc. Financial resources specify the economic aspects of the system. This list of resources is not exhaustive, and new classes can be added to meet the needs of a particular implementation.

Human Resources

The communication model regards systems development as a mainly human activity. Thus, it has explicit representations for the participants and human resources. These two notions allow for easier modelling and automated support for the management of human resources.

Systems involving large and diversified groups of people, must be viewed from technical, social, and psychological perspectives. All three are equally important, and all should be modelled and analyzed. Below, we give an example of a human resources management system. This system maintains the following information about each participant: (1) history of education and work experience, (2) individual career plans and goals, (3) individual preferences in terms of task scheduling (for example: work from home, evenings, part-time), (4) special personality traits, and (5) social role in a team (for example: task initiator, harmonizer, compromiser, or devil's advocate). Thus, the staffing procedure takes into account all aspects of a particular person, not only the technical perspective.

4.5 A Communication Based Description for the Waterfall Model

In this section, we describe a waterfall model using the communication paradigm. We use an exemplary waterfall model from a widely used systems analysis and design textbook [Whitten, Bentley, and Barlow 89]. This particular version displays all general characteristics of a generic waterfall model. The development process is divided into nine phases: survey, study, definition, selection, acquisition, design, construction, delivery, maintenance and improvement. Each phase produces specific documents. The transition between phases depends upon the production of the appropriate documents. Thus, the model is *document-driven*. However, it has a feedback mechanism: a feasibility study conducted after each phase.

For brevity, we model only the first four phases, which traditionally constitute *systems analysis*. Each phase concludes by producing an appropriate document. For the four phases these are a feasibility assessment, a problem statement, a requirements statement, and a systems proposal.

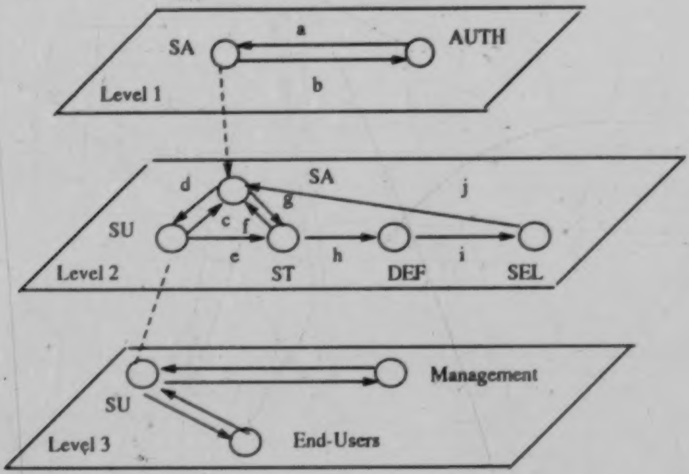

Figure 11: Communication Model for the Systems Analysis

Figure 11 illustrates the communication based representation for systems analysis. Each phase is represented by a *participant*; the documents and related decisions are represented by *messages*. The highest level has two participants: Systems Analysts (SA), and the Authorization Group ($AUTH$).

SA performs system analysis and communicates with $AUTH$ to receive an approval for particular documents. A request for approval is represented by message b; the answer from the authorization group is represented by message a.

Activities performed by participant SA are specified at Level 2, which involves five participants: Systems Analysts (SA), Survey Group (SU), Study Group (ST), Definition Group (DEF), and Selection Group (SEL).

The Survey phase is started by SA, who sends message d to SU. SU sends the results, the *feasibility assessment*, to SA. Then, participant SA performs initial verification and sends the document to $AUTH$. Participant $AUTH$ either approves the documents or sends back a rejection. If the the feasibility assessment is accepted by the users, the process continues. SU sends appropriate documents (message e)

to ST. ST performs activities involved in the study phase (not shown by Figure 11) and sends the results, a *problem statement*, to SA (message f). SA performs the authorization procedure and sends back an authorization or rejection. ST sends the required documents (message h) to DEF. DEF performs the activities related to the user requirements analysis and specification. The results from this phase, a *requirements statement*, are sent to SEL (message i) who, in turn, sends the final document, a formal *system proposal*, to SA (message j). SA sends the system proposal for final authorization by the $AUTH$.

Each activity can be specified in more detail by a lower level description (Level 3). For example, the Survey activities, executed by participant SU, involve End-Users and Management in determining project scope and a preliminary feasibility study. The vertical and horizontal decomposition can be carried on for the remaining participants. We feel, however, that the example above is sufficient to illustrate the following mapping techniques:

1. Phases and activities are represented by the participants at a corresponding level of granularity.
2. Documents and decisions are described as messages.
3. Interactions between participants are defined by the communication protocols.

The representation for the activities corresponds to the real life situation, in which one person or one group is responsible for a particular activity. This person can perform the assigned activity or delegate it to other people or machines; though, one person (or one group) is responsible for the overall results.

In the communication model, development activities can be either processes or tasks (processes with specified desired outcomes). First, we define the activities as processes, then, we create tasks by adding desired outcomes to the processes. Thus, for tasks, the execution of a process is repeated until the particular outcome is achieved. In our example, the survey phase can be repeated until the authorization participant ($AUTH$) sends an approval to participant SA (message a).

A Communication Model for the Module Repository

The above example has illustrated the flow of documents and decisions; however, it has not described how the documents are stored and how the required knowledge is used. Thus, in the following subsection, we decompose the earlier described Survey phase into two processes: knowledge acquisition and documentation storage.

In the example illustrated in Figure 12, we model two instances of knowledge acquisition: (1) initial enquiry about existing standards and (2) addition of application knowledge. The initial enquiry about the project standards is the first step

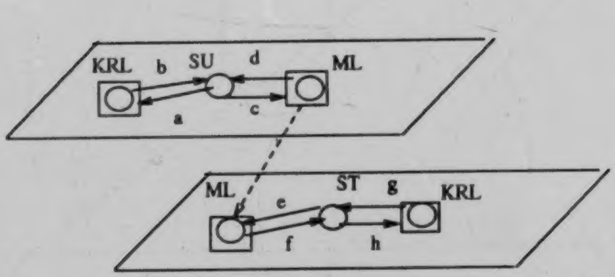

Figure 12: Knowledge and Modules Repositories

of the survey phase (participant SU). As is illustrated by Figure 12, *SU* sends a request for information (message *a*) to the Knowledge Repository Librarian *KRL*. In our graphical notation, a participant is represented by a circle; whereas, a librarian is indicated by a square around a circle. *KRL* performs a search to find all standards related to the Survey Phase as well as the existing examples. When the data is located (or when there is no related information), a response (message *b*) is sent to *SU*.

During the Survey Phase, *SU* gathers the facts about the application. These findings are subsequently added to the Application Knowledge Repository by sending appropriate messages to *KRL*. *KRL* verifies new facts against the existing information and policies. In case of inconsistency, *KRL* sends a notification to participant *SU*, who, in turn, must decide whether the information should be changed or added and marked as inconsistent.

When the required activities are completed, a ***feasibility assessment*** document is sent to *SA* for authorization. Participant *AUTH* performs appropriate activities (not modelled in this example) and sends back either an authorization document or a rejection notice. After the authorization, *SA* sends an appropriate message to *SU*.

If the feasibility study document is authorized, participant *SU* sends the required documents (message *c*) to the Module Repository Librarian (*ML*). *ML* verifies received documents against the existing project standards. The verification of documents is shown in detail at Level 2 (Figure 12). *ML* sends the documents (message *f*) to *ST*. *ST* consults *KRL* (messages *h* and *g*) and sends back the results (message *e*). *ML* either accepts the documents or declines the service. If the answer is positive, *SU* can complete the survey task. However, if the answer is negative, participant *SU* has two options: change the documents to conform to the current standards or change the standards. The process of changing standards depends upon the standard type as well as the policies of the particular project. Thus, we do not elaborate on the change approval, but we describe the change propagation. For simplicity, we assume that a request for a change to the standards has been approved by SA.

A Propagation Task for the Standards Modifications

When the project standards have been modified, the participants of the affected tasks must be notified. The *propagation* task finds all active tasks involving modified standards. The message about the change of standards is sent to all involved tasks (the message is multicasted). This instance of multicasting is an example of a significant reduction in the communications and *information overload*. Generally, large projects have a system in which each change is announced by a memo or notice sent to all team members. As a result, a large number of memos are broadcasted to uninvolved parties. In our example, the information about modifications is sent to interested parties only.

4.5.1 A Communication Based Description for the Spiral Model

The *spiral* model, described by Boehm [Boehm 88], takes a *risk-driven* approach. In this model, the development process evolves through cycles; each cycle involves the same steps. A cycle can represent a phase, an activity, or even the development of a single program. Cycles can exist and evolve in parallel.

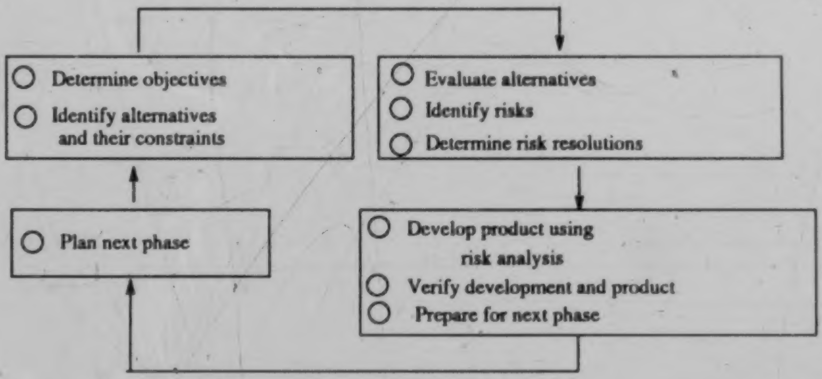
○ Determine objectives
○ Identify alternatives and their constraints
○ Evaluate alternatives
○ Identify risks
○ Determine risk resolutions
○ Plan next phase
○ Develop product using risk analysis
○ Verify development and product
○ Prepare for next phase

Figure 13: A Cycle in the Spiral Model

Figure 13 illustrates the four steps of the spiral model. First step identifies the objectives of the cycle, alternative solutions, and constraints for each alternative. The second step evaluates each alternative in terms of objectives and constraints and identifies risks and their resolution techniques. The next step develops the products taking into consideration the involved risks and their resolution techniques. It also reviews previous steps and products. The last step plans for the next phases.

Risk factors and their possible resolutions are taken into consideration in the development of a particular product. Thus, depending on the type of the risk, for example; financial, health hazard, or user rejection; the development itself can use different methods, such as waterfall, formal transformation, or prototyping.

The mapping from the spiral model to the communication model uses the following methods:

1. A cycle is represented by a high level participant.
2. Each step in the cycle is represented by a lower level participant.
3. Activities performed during each step are represented by the decomposition of participants into a lower level.
4. The results of the steps and the decisions are represented by the messages.
5. The sequence and other interdependencies between activities are defined by the communication protocols.

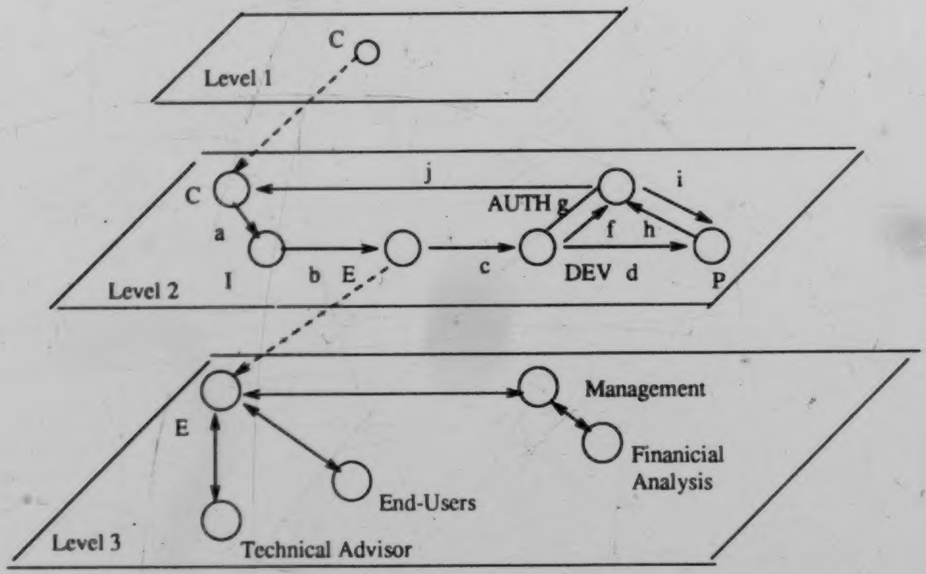

Figure 14: A Communication Based Representation for the Spiral Model

Figure 14 depicts the communication based representation for the spiral model. The highest level participant C represents one cycle in the spiral model. Participants I, E, DEV, and P represent respectively Step 1, Step 2, Step 3, and Step 4.

Participant C starts the cycle by sending message a to participant I. I performs required activities and sends the results (message b) to participant E. E evaluates the alternatives and identifies the overall risks. The evaluation of alternatives and risk analysis are done in consultation with other participants: End-Users, Management, Technical Advisors, and Financial Analysts. Their interactions are shown in Figure 14 at Level 3. The results from the evaluation and risk analysis are sent to participant DEV (message c). DEV develops the system taking into consideration the risks factors. From time to time, DEV consults with $AUTH$ to ensure that the development is carried out applying correct risk resolutions. After the development and verification of the products, participant DEV sends the results (message d) to participant P, who plans for the next phase. P consults with the authorization group, participant $AUTH$. Once the start of the next phase is authorized, participant $AUTH$ sends an authorization message to participant C (message j).

4.5.2 A Communication Based Description for the Prototyping Approach

In this section, we model the *evolutionary prototyping* approach described in Chapter 2. The same mapping techniques can be used for the *rapid throwaway* and *incremental development*. Prototyping techniques basically use the *code-driven* approach, in which the user incrementally specifies requirements while using a succession of prototypes.

Prototyping techniques are represented by two participants: the development team and the users. These participants send messages including prototypes (code) and new or changed requirements. Evolutionary prototyping is depicted by Figure 15. In this example, we concentrate on the specification of requirements. There are two major participants: a systems analyst (SA) and a user (U). The first step

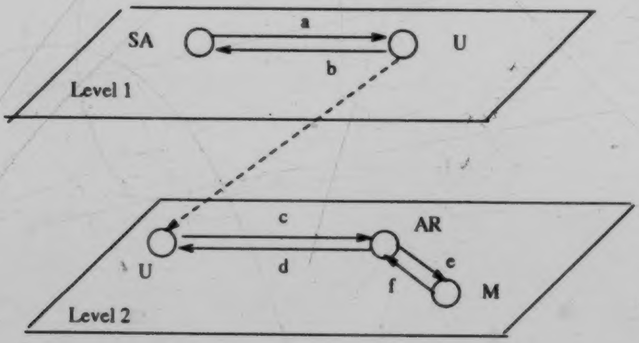

Figure 15: A Communication Based Representation for the Prototyping Technique

is the preliminary analysis and requirements specification. *U* sends initial requirements (message *b*) to *SA*. *SA* then responds with a prototype of the system (message *a*). The user exercises the prototype and responds with requirement changes or new requirements. This cycle of sending prototypes and requirements is repeated until the user sends a message to stop the requirement specification phase. At a lower level (Level 2), the user tests the prototype and sends the results (message *c*) to the requirements analysis group (*AR*). *AR* consults with management, (*M*), and sends message *d* (new requirements, changes, or request for more testing) to *U*.

Modelling of the User Training Process

Gomaa in [Gomaa 83] states that users, particularly novice users, need initial training in the prototyping. Prototypes are difficult to use because they are generally not fully operational and often they do not have sufficient documentation.

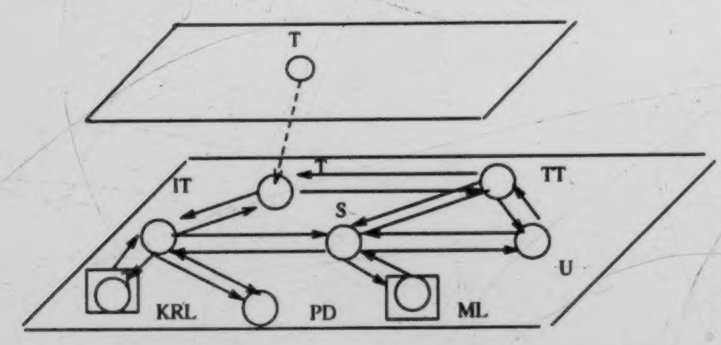

Figure 16: A Communication Based Representation for the Training Process

The training process illustrated in Figure 16 is decomposed into three steps: initial preparation, training session, and testing. It involves eight participants: training group (*T*), initial training group (*IT*), prototype developers (*PD*), session group (*S*), users (*U*), testing group (*TT*), knowledge repository librarian (*KRL*), and module repository librarian (*ML*). In the preparation phase, *IT* sends a request for information pertaining to the particular prototyping technique. The knowledge repository librarian, *KRL*, sends the requested information (message *b*). However, if the information is unavailable or incomplete, participant *IT* consults the prototype developers, participant *PD*. The initial preparation produces two outputs: (1) a training session plan and (2) a training manual (for example: on-line tutorial). These documents are sent to *ML*. The training session involves participant *S* and users, *U*. The last step of training is the testing procedure. *TT* is responsible for conducting appropriate testing to verify the quality of training. The training sessions can be repeated if the test results are not satisfactory.

Chapter 5

Modelling of the Systems Development Activities. A Case Study

5.1 Systems Development Activities

The systems development activities can be classified according to their *duration*, major *driving-force*, and *predictability* (Figure 17).

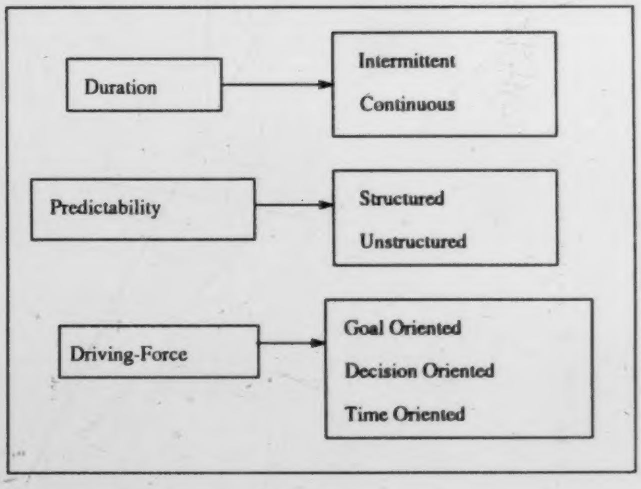
Duration
Intermittent
Continuous
Predictability
Structured
Unstructured
Driving-Force
Goal Oriented
Decision Oriented
Time Oriented

Figure 17: Systems Development Activities

The *duration* of an activity varies from hours to years; some activities are continuous (they last through the entire life of the development process), whereas others are periodical.

A progress of an activity is determined by its *driving force*: *goal*, *decision*, *time*. Thus, we classify activities as mainly: *goal-oriented*, *decision-oriented*, or *time-oriented*. These forces do not exclude each other; in fact, they are often used together. In a system defined in terms of states and transitions, the *driving force* specifies the conditions for a transition from the current state to the next state. Thus, in a goal-oriented process, a transition occurs when a specified goal is achieved, for example, a specific document or code is produced. In the *decision-oriented* process, the progress from state to state depends upon an external decision made by an authorized participant. In the *time oriented* process, the change from state to state is activated by an external clock.

The *predictability* specifies whether the behaviour and the outcome of a process are well known in advance, or are they only roughly determined and incrementally specified.

The currently existing models for the systems development process do not have a uniform representation for all types of development activities. The traditional waterfall model describes a sequence of document-oriented, highly predictable, short and long term activities. It does not describe goal-oriented or decision-oriented activities. The spiral model is a decision-oriented model; however, it does not describe the goal-oriented or time-oriented activities. A model based on the prototyping techniques represents the unpredictable tasks, yet is does not describe decision- and time-oriented activities.

5.1.1 The Communication Model Representation for the Development Activities

The communication model has a uniform representation for all project activities: long and short term, goal-oriented, decision-oriented, time-oriented, predictable and unpredictable.

The *goal-oriented* activities are represented by the tasks, (processes with the specified outcomes). The desired outcomes can be stated as particular products or states, or a combination of both. The task execution is repeated until the goal is achieved. The specification for the *decision-oriented* process involves an authorized participant who decides about the progress of the process. The *time-oriented* process makes the transitions upon the time messages coming from the timer.

The predictable activities are defined as structured processes; the unpredictable as unstructured processes.

5.1.2 Structured and Unstructured Processes

A *structured* process is specified by a set of participants, a protocol, a set of modules, and a set of knowledge repositories. Whereas, the *structured* task involves the structured process description and a set of the desired outcomes. The structured tasks and processes are controlled by their protocols and by an authorized participant, who can decide, independently from the protocol, to suspend, terminate, or abort a process or task execution. Control of the structured task execution is illustrated in Figure 18.

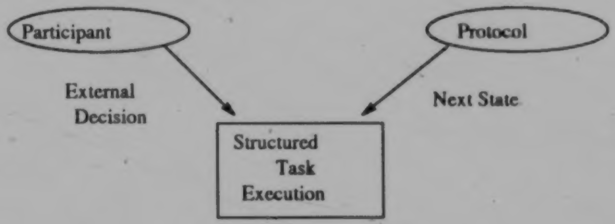
Participant
Protocol
External Decision
Next State
Structured Task Execution

Figure 18: Control of the Structured Task Execution

The *unstructured task* specification has the following components: goal specification, theory description (methods to achieve the goal), accomplishment measurements, and main task and subtasks specifications. An unstructured task is defined incrementally by a number of structured subtasks. The interdependences between subtasks are described by the main task specification.

Control of the unstructured task execution is illustrated in Figure 19.

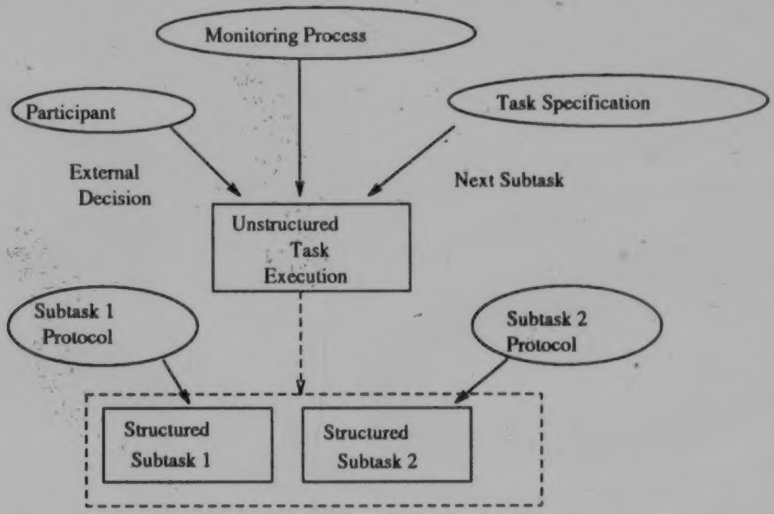
Monitoring Process
Participant
Task Specification
External Decision
Next Subtask
Unstructured Task Execution
Subtask 1 Protocol
Subtask 2 Protocol
Structured Subtask 1
Structured Subtask 2

Figure 19: Control of the Unstructured Task Execution

There are two levels of monitoring mechanisms: (1) the main task controlled by its specification, a monitoring process, and an authorized participant and (2) the subtasks controlled by their respective protocols and authorized participants.

The monitoring process periodically requests the measurements and verifies the state of the overall system. If the status of the process is not satisfactory, the authorized participant is notified or the task is automatically terminated.

5.2 Unstructured Task. A Case Study

A large telecommunication company decided to undertake a special project to protect its revenue by increasing its client satisfaction. Since it had been observed that some of the clients were turning to other telecommunication systems vendors, this company decided to increase the knowledge of the client representatives and provide them with the optimal design tools. The project duration is estimated for three years and the project budget is limited to five million dollars. The project's human resources are limited to the current company's employees. The software and hardware resources can be expanded (by acquisition and developing new products) within budgetary limits.

The following sections provide a definition of the overall task, *Client Service Improvement.*

Goal Description

The goal of the main task is to protect the revenue of a large telecommunication company. Since the revenue is provided by the private customers, the major goal of the company is to increase the number of customers or at least retain the same number. The number of customers and the number and size of their orders are directly related to customer satisfaction.

Theory Description

The task is based on the theory: "Expanding the Technical Knowledge of the customer representatives will increase (or retain) the number of customers."

Accomplishment Measurements

Assuming that the above theory is true, we have to show positive correlation between the knowledge of the customer representatives and the number of customers. Thus, we need two measurements: a level of technical knowledge (competence) and a number of the customers (indicating the customer satisfaction). The evaluation of the customer representatives' knowledge can be done using standardized questionnaires, technical tests, practical exams, or interviews. It will be performed at the start of the task and repeated during the task execution.

The second measurement, a number of customers, must be calculated at the start of the task and then it should be monitored during the task execution. The number of customers can be measured by the raw numbers, or it can be modified by the size of the customers' orders. It can be also adjusted by some other measurements of client satisfaction, for example: by the results from a standardized questionnaire or a structured interview.

In our model, the goal, objectives, and theory (strategy) are clearly stated. Thus, the participants, especially the customer representatives, can be ensured that the particular measurements will not be used for job reviews or work appraisals. The goal of the task is not a selection or elimination of the personnel, but the improvement of the knowledge level and skills of the entire group. Thus, a clear and precise description of the task, which can be accessed and reviewed by the task participants, can significantly lower the level of anxiety among the staff.

Success Condition

The task execution will succeed if, after five months, the technical knowledge of the customer representatives has been increased and the number of customers has been increased or, at least, has stabilized.

Failure Condition

The task should be considered a failure if after five months of a measured increase of the technical knowledge of customer representatives, the number of customers is dropping. A failure of a task execution does not disprove the theory itself; it means that in a given situation using particular measurements, the task has not succeeded in supporting the theory. We must take into consideration the fact that a modelled system represents only a part of the complex environment, leaving out many of the external factors, for example: political situation.

5.2.1 Process Specification

The main task involves four participants: Management Group (MG), Customer Representatives (CR), Technical Support Group (TS), and Project Monitoring Group (PM). The management group has authority to specify, plan, schedule, reschedule, initiate, and terminate tasks. Customer representatives provide the information about the customers (current and potential), and their requirements. Technical support group provides the information about the available tools, and participates in the acquisition and the design of new tools. The project monitoring group makes the periodical measurements.

Since the project activities are highly unpredicatable, the project tasks are specified incrementally. Thus, the main task, *client service improvement*, is specified by a number of structured subtasks.

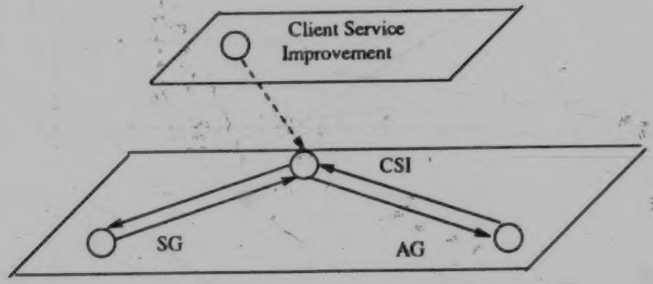

Figure 20: Client Service Improvement Task

In our example, illustrated in Figure 20, the first subtask is defined as a *survey of the current situation*; the second is defined as an *analysis*. The survey subtask is specified first. After its execution, the results are analyzed and the next subtask, analysis, is described.

5.2.2 Task 1. Survey of the Current Situation

Figure 21 illustrates communication between participants in the survey task.

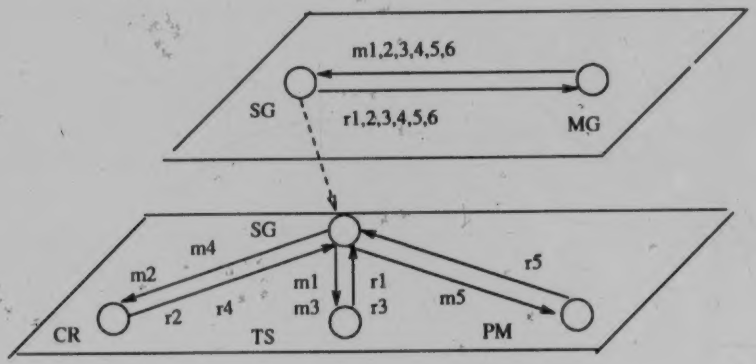

Figure 21: Survey Task Specification

Management group sends messages $m1$, $m2$, $m3$, $m4$, $m5$, and $m6$ requesting six reports:

1. Survey of the tools used by our company (message $r1$),
2. Survey of the tools used by competitors ($r2$),

3. Survey of the commercially available tools ($r3$),
4. Survey of the tools desired by customers ($r4$)
5. Measurements of the customer representatives knowledge ($r5$)
6. The number of customers ($r6$)

5.2.3 Task 2. Analysis of the Survey Reports

Figure 22 illustrates exemplary results from Task 1.

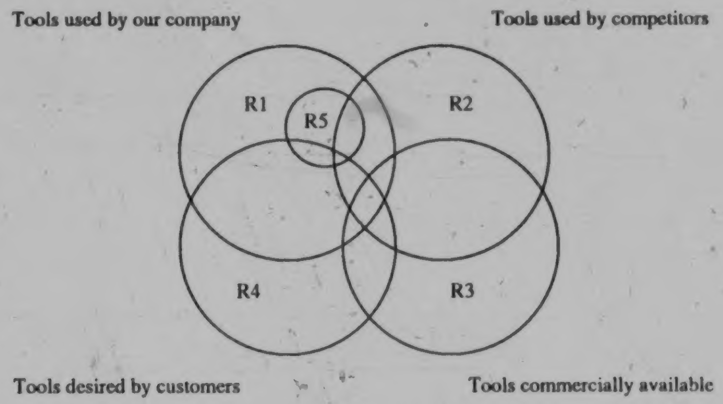
Tools used by our company
Tools used by competitors
R1
R5
R2
R4
R3
Tools desired by customers
Tools commercially available

Figure 22: The Relationships between the Survey Reports

The circles, similar to a Venn diagram, represent the tools described by the corresponding survey reports. Thus, for example, circle $R5$ describes the tools known by the customer representatives. Figure 22 shows that the customer representatives are not aware of many tools offered by our company and desired by the users. Thus, the goal for the next subtask should be: "Increase the customer representatives' knowledge about tools used currently by our company and desired by the customers." Using the symbols from Figure 22, we can describe this goal as increasing the current knowledge ($R5$) to match the intersection between $R1$ (tools used by our company) and $R4$ (tools desired by customers).

Chapter 6

Tasks Specification. A Case Study

Chapter 5 described the systems development activities in terms of their duration, predictability, and driving-force. In the following chapter, we discuss the task specification and the meta-task operations. This chapter is divided into three sections: Communication, Message Definition, and Task Description. The communication section describes the client/server paradigm and the addressing schema used in our model. The second section defines the system messages and gives examples from the Case Study. The last section describes the *meta-task* operations: defining, recording, planning, modifying, scheduling and rescheduling, cancelling, initialization, suspending and restarting, terminating, and aborting.

6.1 Communication

The communication model reflects the technical communication between project team members. In theory, assuming a "democratic system" in communication, a project member can communicate with any other project member. In practice, a project is limited by time and resources, thus communication must be, to some extent, restricted. Consequently, projects use informal and formal protocols to specify who can talk, and when. Similarly, our model uses communication protocols to describe the development activities.

Business communication has three specific characteristics: (1) different communication schemas, (2) adaptability to a particular situation, and (3) unreliability. Since the communication model reflects the nature of human communication, it is concerned with these three aspects.

Communication protocols are capable of representing all inter-group communication schemas [Umstot 87]: interpersonal, serial (a message is passed from one participant to the next one), centralized (one participant servers as a hub in a star-like network), autocratic (one participant sends a message to a small group of participants, then each group participant disseminate the message to subordinate groups), and decentralized (message is passed in a circle).

Protocol specification includes many optional paths; thus, the execution of a protocol varies depending upon circumstances. For instance, a protocol can have two communication paths: emergency and normal. In an emergency situation, the protocol is executed with reduced communication (for example: certain authorization procedures can be bypassed). Whereas, the same protocol, executed under normal circumstances, involves all specified communication interactions.

Because of the human participation (people often forget to reply or attend the meetings), communication between participants is unreliable. Therefore, the communication model requires a *time-out* mechanism to improve reliability of the system. The timing issues will be discussed later, in a context of a particular formalism.

To provide the required flexibility and universality, our model uses asynchronous communication, three addressing schemas, port-to-port addressing method, and client/server paradigm.

6.1.1 Asynchronous Communication

The *asynchronous* mode of communication is based on the concept of a *mailbox*. Thus, messages are sent to mailboxes and placed according to their priorities. Participants check ~~their~~ mailboxes and perform appropriate actions. The asynchronous mode of communication allows participants to perform other operations while waiting for the reply from a server.

The communication model uses the asynchronous communication for two reasons. First, this mode reflects business communication between the project members, in which memos, and notes are placed in the mailboxes, the phone calls are recorded, and meetings and appointments are arranged by a secretary. Second, asynchronous communication is more general than synchronous; it is able to represent synchronous communication, but the opposite is not true.

6.1.2 Three Addressing Schemas

The communication model uses three addressing schemas: *point-to-point*, *multicasting*, and *broadcasting*. *Point-to-point* involves one sender and one receiver; multicasting corresponds to one to a group communication; broadcasting involves one to everybody communication. These schemas allow for representing different types of communications: interpersonal (point-to-point), serial (point-to-point), centralized (broadcasting), autocratic (multicasting), and decentralized (point-to-point and multicasting).

6.1.3 Port-to-Port Addressing Method

Messages are sent from the client port to the service port. Ports are the access points to appropriate services. A service can be supported by a number of servers or it can be executed by the same participant. A *port* is a communication channel — a priority queue protected by the system. Port-to-port addressing provides a uniform access to all local and remote objects, which allows modelling of a distributed system.

6.1.4 Client/Server Paradigm

The communication model uses the *client/server* paradigm, in which a *client* participant sends a request to specific *server* participant. Upon arrival, the service request is placed in a priority queue. When the request is received, the server performs requested operation and sends a reply to the client.

6.2 Message Definition

Message is defined as a typed collection of data objects used in communication between participant. The particular implementation of the communication model can employ different classes of messages. For instance, it can have two types of messages: *short messages* and *messages with data segments.* The data segment may include the actual data or the address and the access rights. Thus, data can be passed *by value* or *by reference.* In the first case, the data segment contains the actual data; for example: a task description or copy of a project module. In the second case, the data segment has only the data address; for example: the task identifier or project module identifier.

A message has two groups of information: *message related data* and *task related data.* The message related information has five components: identification, authorization, target, priority, and reply specification. The task related information has four components: authorization, priority, meta-task operation, and task description.

```
message ::= <message data, task data>
message data::= <identification, authorization, target, priority,
                reply specification>
task data::= <task authorization, task priority, meta-task operation,
              task description>
```

6.2.1 Message Related Data

Message identification is a unique name, a tag which distinguishes messages. It may be implemented, for example, as a combination of a number, a date-time stamp, or a sender identifier.

```
message identification :: = <date-time stamp, sender_id>
```

Message authorization identifies the sender as a person authorized to send this message type and also relates this message to a higher level task description.

Message target is a specific port address. A message can be sent only to a port with known address. The methods of disseminating the information about addresses are specific to a particular implementation. For instance, some ports can have public addresses — known to all participants, while others can have protected addresses, known only to a group of participants.

Message priority specifies the order in which messages are received from the mailbox. There should be a special provision made for emergency messages. For example, an emergency message "Abort task" should force the server to immediately stop the task.

The *reply specification* describes the reply protocol. For instance, it can be defined in one of the following ways:

1. Send acknowledgement when the message is received and send reply upon completion of the task.

2. Do not send acknowledgement but send status information every 24 hours until the task is completed.

3. Send reply only upon completion of the requested task.

4. Do not reply.

6.2.2 Task Related Data

Task authorization provides the data relating this task to other tasks in the system. These relationships are used by the explanatory mechanism, which gives answers to questions: "Why this task is being executed?", "What would happen if the task is cancelled?."

Task priority can be expressed in term of arbitrary priority numbers or in terms of a due date.

Meta-Task Operation specifies an operation which will be performed upon a task specification, task execution or the history of an execution. They are described in section 6.3.

6.2.3 Case Study. Messages for Task 1

The following section describes the messages for Task 1 — Survey of the Current Situation (Chapter 5, Figure 20).

The Management Group sends message 1 (*m1*), requesting the information about the design tools offered by our company. The message is sent to the Technical Support Group. A reply is expected in two weeks in the form of a Survey Report (message *r*1).

```
-----------------------------------------------------------------

    message id:  m1
        sender:  Management Group
 authorization:  Survey of the Current Situation - Request No. 99
        target:  Technical Support Group
      priority:  high

  Reply specification:  upon completion , within 2 weeks from
                        the receiving date.

  Task Specification: goal:  the design tools used by our company
                    method:  all decisions delegated to the
                             message target
                    result:  Survey Report (Subtask1.r1)
                  priority:  high
-----------------------------------------------------------------
```

The Management Group sends message 2 (*m2*) to the Customer Representatives Group, requesting the information about the design tools used by the competitors. sent to the Customer Representatives Group. A reply is expected in two weeks in the form of a Survey Report (message *r*2).

```
--------------------------------------------------------------------

    message id:  m2
        sender:  Management Group
 authorization:  Survey of the Current Situation - Request No. 99
        target:  Customer Representatives Group
      priority:  high

  Reply specification:  Acknowledge upon receiving,
                        Reply upon completion, within 2 weeks
                        from the receiving date.
```

```
Task Specification: goal: the design tools used by the
                          competitors
                    method: all decisions delegated to the
                          message target
                    result: Survey Report (Subtask1.r2)
                  priority: high
----------------------------------------------------------------
```

Message 3 (*m*3), requesting the information about the design tools commercially available, is sent to the Technical Support Group. A reply is expected in three weeks in the form of a Study Report (message *r*3). Message 4 (*m4*), requesting the information about the design tools desired by the customer, is sent to the Customer Representatives Group. A reply is expected in three weeks in the form of a Requirements Report (message *r*4). Message 5 (*m5*), requesting the current measurements of the customer representatives knowledge, is sent to the Monitoring Group. A reply is expected in one week in the form of a Measurements Report (*r*5).

6.3 Meta-Task Operations

Tasks and processes are defined and modified by participants. The standard types of meta-task operations are: defining, recording, planning, modifying, scheduling and rescheduling, cancelling, initialization, suspending and restarting, terminating, aborting, and accessing.

Figure 23 illustrates the operations on a task specification, a scheduled task, a task execution, and a task execution history.

Defining, *recording*, *planning*, *modifying*, and *scheduling* refer to task specification. A scheduled task can be *rescheduled*, *cancelled*, *modified*, or *initialized*. The task execution can be *suspended*, *terminated*, or *aborted*. The suspended task execution can be *restarted* and the history of task execution can be *accessed*.

Task Defining

Similarly to other types of software, tasks specifications can be shared between projects, acquired from vendors, or written by participants for a specific project. Participants prepare new tasks specifications using the *task defining* process, which has, for example, the following steps:

1. Identification of related information: (1) similar tasks (their reusable components) and (2) project policies regarding similar tasks

2. task specification, which may involve consultations with other participants

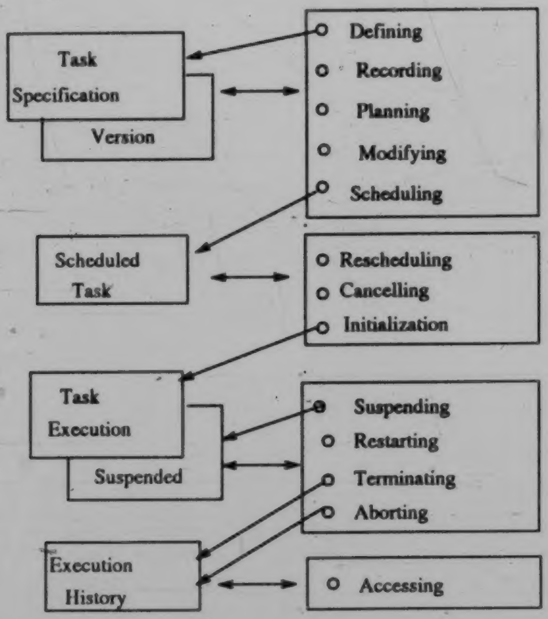
Task
Specification
Version
Defining
Recording
Planning
Modifying
Scheduling
Scheduled
Task
Rescheduling
Cancelling
Initialization
Task
Execution
Suspended
Suspending
Restarting
Terminating
Aborting
Execution
History
Accessing

Figure 23: Meta-task Operations

3. Verification of the task specification against project policies.

In the first subtask, the participant sends a request to the Knowledge Repository Librarian. The librarian finds the related information and sends it back to the requesting client. In the second subtask, the participant decides which tasks can be reused, consults with other participants, and requests further information from the librarian.

Example of the Initial Task Specification

The following example describes the defining process for the task providing the current measurements of the customer representatives knowledge (Report 5).

```
Task Decomposition:

  1. Prepare tests for the customer representatives.
  2. Prepare interview specification.
  3. Perform test.
  4. Conduct interviews.
  5. Evaluate test results.
  6. Prepare final report - R5.
```

The knowledge repository librarian sends the following information about related tasks and project policies:

1. Skills and knowledge evaluation uses two methods:
    (1) Objective test (theoretical and practical) administered to all test subjects at the same time.
    (2) Individual interview with the test subject, which indicates the skills and knowledge as well as the subject's attitudes and concerns.
2. Histories of previous tests indicate that the employees are accustomed to the test situations.

Related Policies:

1. Evaluation of employee's skills or knowledge must be preceded by a notice with an explanation of the purpose of the evaluation.
2. The evaluation results must be kept confidential.

Taking into account the related policies and standards, the task decomposition must be modified in the following way:

Revised Task Decomposition (revisions are indicated by arrows):

--> 1. Obtain Report 2 (Survey of Tools used by our Company) and prepare material for testing.
--> 2. Prepare a notice explaining the purpose of this evaluation. Ensure the customer representatives about the confidentiality of the test results.
3. Prepare tests for the customer representatives.
4. Prepare interview specification.
5. Perform test.
6. Conduct interviews.
--> 7. Evaluate test results (ensure the confidentiality).
--> 8. Prepare final report - R5 (ensure the confidentiality).

Task Recording

The *Recording Task* sends a request to the Knowledge Repository Librarian requesting the addition of a new task specification. Librarian verifies the project policies and stores the specification.

Task Planning

The *Planning Task* performs a tentative resource allocation: human, software, hardware, equipment, finances, and time. It verifies whether the task is possible with the existing resources. If the project resources are not sufficient, the planning task may start a resource acquisition task or it may send back a reply outlining the existing and required resources. Planning may also include the task analysis procedure: bottleneck analysis, overload and underload analysis (for the tentatively allocated resources).

Task Scheduling and Rescheduling

The *Scheduling task* executes the resource allocation procedure for a specified time or first available time. In the first case, when the time is specified, as for example next week, the allocation process finds all required resources available next week. If the resources are available, the server sends a positive reply to the client. The particular details of the allocation schema are discussed in Appendix A.

When the resources are unavailable, the resource allocation performs the priority verification task. The Priority Verification Task checks the priority of the tasks allocated to the resources. If the priority is lower than the current task, a task rescheduling can be started.

After the task is placed in the scheduler queue, it can still be rescheduled or cancelled. Tasks can be rescheduled as the result of higher priority tasks or unavailability of resources, or an explicit request from the client.

Task Cancellation

The *Task Cancellation* process removes the scheduled task from the scheduler queue and releases allocated resources. The cancellation may or may not be recorded, depending on a particular project policy.

Task Initialization

The task is initialized by two methods: (1) automatically by the scheduler at a specified time and under specified conditions or (2) manually by an external participant.

Task Modification

The task specification and task execution can be modified. However, in both cases a specific changing mechanism is required. Thus, in case of specification change, the current executions of the specification must be suspended and restarted with the new version of specification. In case of execution modification, the execution must be suspended, the task specification modified, and then the task execution resumed. A specific implementation may involve different change mechanisms.

Task Suspending and Restarting

A task execution can be suspended and later resumed. However, this operation stops the task execution for a short time without releasing resources. Otherwise, the resources must be released and the remaining subtasks must be rescheduled and then restarted.

Task Terminating

The execution of a task can be terminated upon a request from the authorized participant. Termination is performed according to its protocol. For instance, it may require completion of a special report or a subtask. Thus, the terminated task does not stop immediately.

Task Aborting

Task execution can be aborted by an authorized participant. In this case, all task actions stop immediately.

Accessing the Execution History

Since the history of each execution is automatically stored, it can be accessed by the authorized participants. A history contains also, in case of terminated or aborted tasks, an explanation for these actions.

Chapter 7

Formal Specification of Systems

This chapter is concerned with the system specification. The first section describes three levels and domains of the specification. The *abstract*, *dynamic*, and *complex* nature of specification is described in Section 7.2. The subsequent section examines relationship between the specification and its implementation. Section 7.4 defines the term *system specification*. The last two sections describe, respectively, formal specification and the difference between verification and validation.

7.1 Levels and Domains of Specification

In our work, we use the term *specification* at three levels: (1) systems development process, (2) system, and (3) module. Furthermore, we refer to three domains: (1) application (specified system), (2) abstract model, and (3) implementation.

Chapter 1 described two distinct, yet closely related, levels of specification: (1) the software systems development process and (2) software system. Now, we are introducing a third one: *local specification*. The distinction between the *local* and the *system* specification, introduced by Guttag in [Guttag 81], is pivotal for the development of the appropriate specification methods. On the other hand, we have argued in Chapter 1 and 2 that the essential attributes of the software systems development process and the software system are similar. Therefore, in this chapter, we refer to both as *system specification* and, as a result, we discuss only two levels of specifications: *system* and *local*.

The *local* specification applies to a single program unit, and its scope is limited to the module and its interface with the other modules in the system. In most cases, local specification is created and used exclusively by the system developers.

The *system* specification differs significantly in its scope and usage from the local specification. Its scope is to describe the entire system and its interface with the environment. Furthermore, system specification describes not only the system behaviour, but also the system attributes, such as: performance, security, reliability, robustness, or viability. In addition, it is created and used by diversified groups of people: managers, systems analysts, and programmers.

Although the system specification is far more complex than the local specification, the literature on formal specification either concerns itself exclusively with the local specification, or does not recognize the difference between them. A direct transfer of the methods used in the local specification to the system specification creates two problems. First, many specification techniques and languages, used for a single module specification, do not scale up to a large and complex system. Second, most local specifications use specialized languages based on advanced mathematical concepts, which are not accessible by the heterogeneous group of people participating in the system specification.

7.2 Characteristics of the Specification

7.2.1 Abstract and Linguistic Nature of Specification

Both the *system* and the *local* specification are abstract linguistic constructs, which are used in human-machine and human-to-human communications. Therefore, specifications must be machine and human readable, or they must be automatically translated into human readable texts. As a result, specifications use a combination of natural, artificial, and pictorial languages (graphical specification).

7.2.2 Dynamic Nature of Specification

Specification is an intrinsic part of the entire software development process and it has an evolutionary and dynamic character. Thus, the system specification must be viewed as a process, not as a final product.

The traditional *waterfall* model assumes a linear sequence of the analysis and design steps, in which the specification has to be completed before the design can be started. In reality, the analysis and design process has an iterative and evolving nature.

As it was noted by Floyd in [Floyd 89], the system specification is not predefined, it is rather created throughout a number of interactions between the users and the system developers. In this view, the specification process is a learning process. The users have to learn a new perspective on their system, whereas the developers learn about the organization of the existing system and its goals.

Furthermore, specification must reflect changes in the modelled application and its environment. Since systems and their environments do change, the system specification is never finished and is constantly under a number of revisions.

7.2.3 Complex Nature of Specification

System specification is complex because (1) it describes many aspects of a system and (2) the systems it specifies are often complex.

Specification describes both the *functional* and *non-functional* perspective on a system. Functional specification defines the system from the operational perspective. The non-functional specification describes attributes such as performance, security, reliability, robustness, and viability. However, depending on a specific system, the non-functional attributes can be also functional. Moreover, some aspects, such as ergonomics (for example user interface), maintainability, or reusability; are difficult to classify. In our work, we concentrate on the functional description of a system.

In addition, specification reflects the complexity of the specified system: its size and a number of relationships between its components.

Decomposition and abstraction are the two methods used to decrease the complexity. A modular approach allows for decomposition of the specification into smaller, more manageable, parts. An abstraction method allows for constructing multilayered specification, in which the top layer includes the requirements specification and each consecutive layer has more implementation details. As a result, the system specification must include mappings from the higher to the lower levels.

7.3 Intertwining of the System Specification and Implementation

Ideally, the conceptual specification should be independent from the physical aspects of the system. Abstract description allows for creation of a specification with many possible implementations and it eliminates the reexamination of the conceptual layer after each change in the physical layer. However, in the case of the software systems it is difficult to describe precisely what constitutes the specification of the system and what constitutes the system implementation. The distinction between them is based only upon the level of abstraction. Moreover, some aspects of the system specification, for example, the time- and security-related specifications, depend on the physical implementation of the system. However, relatively few works recognize the intertwining of specification and implementation ([Swartout and Balzer 1982], [Floyd 1989]).

7.4 Specification Definition

The term *system specification* is used here to denote a specification (or a set of specifications) of a software system. The software system is described, for the purpose of this work, as an open system implemented totally or partially by a set of computer programs.

The conceptual specification is a description of the users' requirements mapped to a specific conceptual model and expressed in a particular language. The implementation specification is a mapping from the conceptual specification into the particular implementation of the system.

To generalize, we define specification as a sequence of layers S_i. Each layer is a text expressed in a particular language (or languages) and referring to a specified model (or models). Languages can be graphical or sequential and formal or informal (natural language). Models can be represented by formal systems or general descriptions.

One specification layer should correspond to one level of abstraction. This is usually guaranteed by the model and language used at a particular level.

Specification also includes the mappings between the layers. Mapping M_i describes (1) the methods creating layer $i+1$ from layer i and (2) the verification methods to demonstrate that the requirements from a higher level are preserved by the lower level.

The specification at level i is a tuple:

$$S_i = < T_i, L_i, V_i, M_i >$$

where
T_i is the set of specification texts for layer i
L_i is a set of languages used at layer i
V_i is a set of models used at layer i
M_i is the set of mapping functions from layer $i-1$ to layer i and reverse functions or verification methods from layer i to layer $i-1$.

Specifications use three classes of semantics: denotational, operational, and axiomatic. Other authors described these methodologies as two approaches - restrictive and prescriptive [Lamport 86], or as two styies - definitional and operational [Liskov and Guttag 86]. The *prescriptive* approach gives a high-level description of all possible behaviours of the designed system. An implementation is equivalent to the specification, which means that all specified behaviours have their representations in the implementation. The restrictive approach specifies a system by a set of properties it must satisfy. It allows a construction of different implementations exhibiting different behaviours as long as all of them satisfy specified properties.

7.5 Formal Specification

The specification methods may be classified as *informal*, *semiformal* (rigorous), and *formal*.

The specification is formal when all components are formal:

1. The languages L_i are formal (they have formally defined syntax, semantics, and pragmatics)
2. The models V_i are formal
3. The mappings between layers and verification methods are formalized.

Formal in this context refers to having mathematical basis. In more restricted meaning, formal description implies that all properties of a system can be proved.

7.6 Verification and Validation

Some authors use the terms *verification* and *validation* interchangeably ([Bochmann 87]) while others use them with different meanings. For example, Rudin ([Rudin 87]) uses *validation* to describe checking of the syntactical correctness and *verification* to analyze the functional aspects of a system. However, from the software engineering perspective verification and validation have two different scopes; verification is, in fact, a part of validation. Verification determines the correspondence between a software system and its specification. Also, it demonstrates consistency, completeness, and correctness of software. The scope of validation is broader and it includes the evaluation of the system by its users. Boehm in [Boehm 81] defines the verification and validation in the following way:

> Verification: To establish the truth of correspondence between a software product and its specification (from the Latin *veritas*, "truth"). Validation: To establish the fitness or worth of a software product for its operational mission (from the Latin *valere*, "to be worth").

In our work, we use terms *verification* and *validation* in the above sense.

Chapter 8

Communicating Abstract Machines as a Formal Specification for the Communication Model

8.1 Communication System Modelling

Reactive Systems

Our communication model belongs to the broad class of *reactive systems.* The term *reactive system* emphasizes that the major purpose of the system is interaction with its environment. This is different from the sequential approach, in which the interactions take place exactly twice: at the beginning and upon termination of the process. For instance, sequential programs can be viewed as functions from the initial state to the finial ~~st~~ate, which accept input values at the beginning and produce output values upon termination. The traditional *waterfall* model is an example of the sequential approach to the systems development process.

A formal specification of a reactive system is difficult, since (1) reactive system interacts with the environment throughout its execution, (2) the execution is usually long-term or non-terminating, and (3) reactive system is often distributed, concurrent, non-deterministic, and real-time. On the other hand, the formal specification is necessary to manage the complexity of this class of systems.

Communication System Definition

A specification for the communication system includes (1) definition of the interface, (2) description of processes, (3) description of interprocess communication, and (4) behavioral constraints. Hence, we define communication system S as a quadruple:

$$S = < E, \{M_i\}_{i=1}^{n}, C, T >$$

where E is a description of the interface with environment, M_i is a formal description of process P_i, C is a specification for the interprocess communication, and T defines constraints on the behaviour of the system: start, termination, or failure condition, for example.

In this chapter, we concentrate on process specification since other components are implementation dependent. To describe processes, we introduce a formalism based on the concepts of *states* and *transitions* — *Communicating Abstract Machines.* This formalism is only one of the possible descriptions, and a particular implementation of our model can be based on any other formal technique, for instance: Petri Nets ([Peterson 81], [Liu and Horowitz 89]), the Calculus of Communication Systems (CCS) [Milner 80], Communicating Sequential Processes (CSP) [Hoare 85], Temporal Logic [Lamport 80], or Actors [Agha 86].

Our choice of a formal language is based on three premises: (1) Formalism for the communication model should have *operational* semantics; (2) It should be accessible to diversified groups of people; and (3) It should be based on one of the existing techniques. The state-based approach meets all three requirements. Its semantics is operational; therefore, it can be easily visualized and enacted by the developers ([Shwarz 82], [Pattavina 84], [Joseps 88], [Lam and Shankar 90]). It is widely used in software engineering and many other fields. Furthermore, its graphical representations, such as State Transition Diagrams and Statecharts, are supported by many CASE tools.

On the other hand, methods based on temporal logic or algebra, Milner's Calculus of Communication Systems [Milner 80], for example, use a specialized notation and reasoning methods which restrict their usage mostly to academic and research settings [Lam and Shankar 90].

Applications of State Machines

Various fields use the state machine concept: mathematics (automata theory), computing science (complexity theory, language recognizers), linguistics (Augmented Transition Networks), engineering (digital circuits design), and systems modelling. This fact demonstrates that the state-based formalism is widely accepted and successfully used by diversified groups of people. In the context of the communication model, we are interested, especially, in two applications: computer networks and systems analysis and design.

In computer networks, the FSMs are used to model communication protocols. In particular, *Estelle* (Extended State Transition Language) is a popular formal description standard accepted by ISO. The theoretical basis and practical implementations of Estelle are described in ([Merlin 75], [Merlin 76], [Lam and Shankar 84], [Jard 85], [Kovacs 86], [Lombardo 86], [Palmer and Sabnani 86], [Ross 87], [Bohmann 77, 78, 80a, 80b, 82, 87] [Budkowski and Dembinski 87]). The formal specification language Estelle was extended by the notion of time [Dembinski and Budkowski 87] and supported by a number of software tools [Ansart 86], among them C and Smalltalk compilers and a dynamic graphical representation GROPE [New and Amer 89].

In systems analysis and design, state machines are used to model system processes. The standard techniques include State Transition Diagrams, PERT charts, extended Data Flow Diagrams, and Statecharts. For example, Ward, in [Ward 86], added state machines to the traditional DFDs. These automata, called *transformation schema*, define timing and control, whereas DFDs describe the overall flow of information. STATEMATE is a graphically oriented system based on Harel's *Statecharts* and used for the specification, analysis, design, and documentation of reactive systems ([Harel 87], [Harel 88], [Harel et al. 90]),

Since state machines are used in various disciplines, each field has developed its own terminology and notation. In addition, the original FSMs were extended in many ways to include: state variables, enabling predicates, procedures, timing, probabilities, and structural decomposition. Therefore, to describe the *Communicating Abstract Machines*, we must first define the basic terms and give a short history of the extensions to FSMs.

Limitations of Simple FSM and their Solutions

The original Finite State Machines were created to describe small systems. As a result, when they have been used for the specification of large reactive systems, the developers encountered many problems. Thus, the simple state machines have been extended in many ways to overcome the "traditional" limitations. Four problems are discussed in the literature [Harel 87]: (1) "Flat" and unstructured nature of FSMs; (2) *State explosion*; (3) A separate representation for each transition; and (4) The sequential nature of FSMs. Figure 24 illustrates the four problems and their solutions.

Organization of this chapter follows the history of FSMs. Sections 8.2, 8.3, and 8.4 describe the Finite State Machine models used for the specification of communication protocols: *Simple Finite State Machine*, *Abstract Machine* (Extended Finite State Machine), *Communicating Finite State Machines*, and Extended State Transition Language (Estelle). Section 8.5 discusses representation methods for FSMs. Sections 8.6 and 8.7 define *Communicating Abstract Machines* (CAM) and provide specification for Subtask 5 from case study. Section 8.8 adds the notion of time to the CAM and gives an example of time specification for Subtask 5.

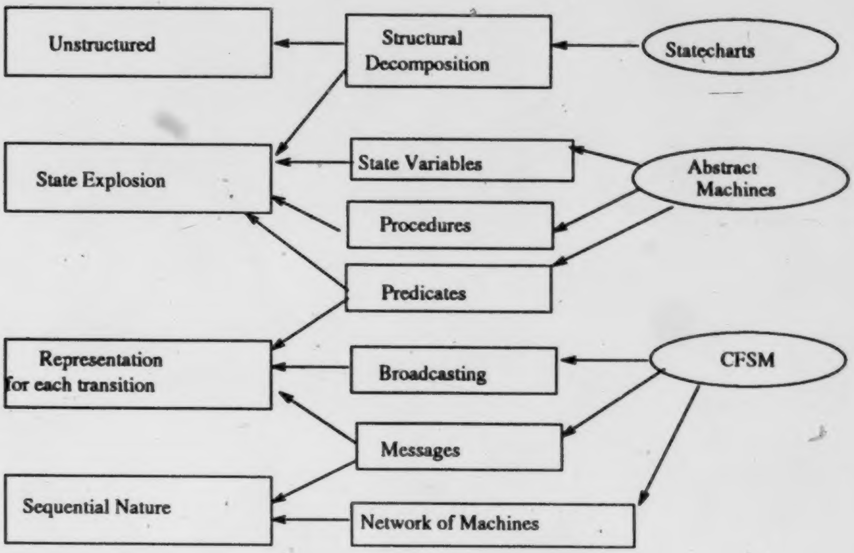
Unstructured
Structural Decomposition
Statecharts
State Explosion
State Variables
Abstract Machines
Procedures
Predicates
Representation for each transition
Broadcasting
CFSM
Messages
Sequential Nature
Network of Machines

Figure 24: Limitations of simple FSMs and their Solutions

8.2 Simple Finite State Machine

The articles by Bjorner, Bochmann, and Danthine ([Bjorner 70], [Bochmann 78], [Danthine 80]) were among the first works adopting simple Finite State Machines to model communication protocols. Their articles were concerned mainly with the following issues: discrete character of FSM, sequencing of the events, and synchronization.

Discrete Character of the FSM

The Finite State Machine models are characterized by discrete-valued inputs, outputs, and internal elements; whereas, the continuous machines can accept and provide continuous signals. On one hand the discrete model of processing is limited by the virtue of digitalization, but on the other hand it better reflects the nature of the digital computer. Since the FSM has a discrete nature, modelling of real-time systems requires methods for time quantization.

Sequencing of the Events

A basic Finite State Automaton gives an output value depending on the current input and a sequence of previous inputs. Each sequence of inputs is named by a *state name*. A process modelled by a single FSM can only accept one input and produce one output at a time. It can be only in one state (called the current or present state) and perform one transition at a time. During its execution a machine can be either in a state or a transition. Since the finite state machine is sequential, parallel execution must be represented by a set (network) of the single machines.

Synchronization

There are two types of finite state machines: *synchronous* and *asynchronous*. A synchronous machine works upon a clock pulse; each transition is executed upon a clock tick. Time itself is modelled as an ordered set of integers. Thus, at time t, the machine is in state $q(t)$ and and at time $t+1$ it is in state $q(t+1)$. An asynchronous machine executes its transition upon occurrence of a triggering event.

8.2.1 Deterministic Finite State Machine Definition

The following definition is based on the Finite State Automaton described by Danthine, in [Danthine 80].

A *deterministic finite state machine* is a tuple :

$$FSM = < Q, \Sigma, q_0, f, g >$$

Q - a finite set of states
$\Sigma = I \cup O$ - a finite set called input/output alphabet
q_0 - the initial state, $q_0 \in Q$
f - a total function called the state transition function $f : I \times Q \rightarrow Q$
g - a total function called the output function $g : Q \times I \rightarrow O$

The g function corresponds to a Mealy model of the FSM, in which the output depends on both the present state and the present input. The other possibility for the output function is the Moore model, in which the output is determined only by the present state, so the output function is defined as $g : Q \rightarrow Q$. In the next sections, we refer to the Mealy model of FSM.

Transition Function

Automata perform one or both of the following tasks. They compute partial functions $X^* \rightarrow Y^*$ (X and Y are finite alphabets) or they recognize languages $L(X)$ over alphabet X.

The λ transition is introduced to represent an internal event of the machine which is not triggered by the input event. Lambda transitions are used to compose a complex machine from a number of submachines.

Deterministic finite state machine with λ-transitions is defined as a tuple :

$$FSM - \lambda = < Q, \Sigma \cup \{\lambda\}, q_0, f, g >$$

Q - a finite set of states
$\Sigma = I \cup O$ - a finite set called input/output alphabet
q_0 - the initial state, $q_0 \in Q$
f - a total function called the state transition function $f : I \cup \{\lambda\} \times Q \rightarrow Q$
g - a total function called the output function $g : Q \times I \cup \{\lambda\} \rightarrow O$

8.2.2 Nondeterministic Finite State Machine

In deterministic machines, the next state is completely defined by the present state and input; in nondeterministic machines a number of next states can be possible. Nondeterminism can also be added by introducing a set of initial states instead of one initial state.

A *nondeterministic finite state machine* is defined by a tuple:

$$NFSM =< Q, \Sigma, Q_0, f, g >$$

Q - a finite set of states , $\mathcal{P}(Q)$ - a power set over Q
Σ - Input/output alphabet
Q_0 - a set of initial states, $Q_0 \subseteq Q$
f - a total function (transition function) $f : I \times Q \rightarrow \mathcal{P}(Q)$
g - a total function (output function) $g : Q \times I \rightarrow \mathcal{P}(\mathcal{O})$

8.3 Abstract Machine (Extended Finite State Machine)

In the modelling of more complex systems, the number of states can easily become immense. This limitation of the FSM is known as the *state space explosion* problem. To eliminate the necessity of state enumeration, an abstract model was proposed with state classes and state variables. This extension was originally done by Keller [Keller 76] for parallel programs. Later, abstract machines were used in modelling of communication protocols ([Bochmann and Gecsei 77], [Danthine 80], [Bochmann and Sunshine 80], [Simon 82], [Bochmann et al. 82], [Linn 88]).

The abstract model introduces *state space variables* and extends transitions by *enabling predicates* and *procedures* written in programming languages.

The states are organized into classes and FSM is extended by a vector of local variables $V = (v_1, v_2, ..., v_i)$. A state class type is described as a tuple $< s, v_i >$, where $s \in S$ is a type name, and v_i is a vector of variables, called *state variables*. The instances of the states are described by the type name and the values of the state variables. If the variables are unbounded, the number of state instances can be infinite. Therefore, the name *abstract machine* seems to be more appropriate than *extended finite state machine.*

The transition is extended by an *enabling predicate* P_t and an *action function* F_t. An enabling predicate is a logical statement on the state variables and the event value. When the transition is triggered by the appropriate event or set of events, the enabling predicate is evaluated. If the predicate evaluates to a true, the corresponding function F_t is calculated, the next event or events are generated, and the machine changes its state to the next one. Function $F_t(v_i)$ is described by a procedure written in one of the programming languages, for example: Pascal, C, or Smalltalk.

Thus, transition t from state q_i to state q_{i+1} is represented by the following tuple:

$$t =< q_i, E_t, P_t(v_i), F_t(v_i), q_{i+1}, E_n >$$

where
$q_i, q_{i+1} \in Q$ and q_i is the present state, q_{i+1} is the next state
$E_t \subseteq \mathcal{P}(\Sigma)$, set of triggering events
$E_n \subseteq \mathcal{P}(\Sigma)$, set of output events. $P_t(v_i)$ - an enabling predicate
$F_t(v_i)$ - transition function.

Abstract Machine Definition

The *abstract machine* is defined as the following tuple:

$$AFSM =< Q, \Sigma, Q_0, \psi, P(V), V >$$

Q - a state space, $Q \subset S \times V$
Σ - a set of events
Q_0 - a set of initial states, $Q_0 \subset Q$
ψ - a mapping $Q \times \Sigma \times P(V) \rightarrow Q \times \Sigma \times F(V)$
V - a vector of local variables
$P(V)$ - a set of predicates over V
$F(V)$ - a function $V \rightarrow V$

The system behaviour is described by the set of global states and transitions between them. A global state is a pair $< S, E >$ where S is n-tuple $< s_1, s_2, ..., s_i >$ of entity states and E is n-tuple of the triggering events lists.

8.4 Communicating Finite State Machines

Aho, Ullman, and Yannakalis are among the first to use the communicating state machines for the protocol specification [Aho et al. 79]. Later, the *Communicating*

Finite State Machines (CFSM) were described in more detail Brand and Zafiropulo [Brand and Zafiropulo 83] and Lam and Shankar [Lam and Shankar 84].

In the model described by Brand and Zafiropulo, the mechanism enabling the transitions is based on sending and receiving messages. The entities in a communication system are modelled as finite state machines, processes are described by *protocols*, and the interprocess communication is specified by channels with messages.

Brand and Zafiropulo define *protocol* as the following tuple:

$$<< S_i >_{i=1}^{N}, < o_i >_{i=1}^{N}, < M_{ij} >_{i,j=1}^{N}, succ >$$

Where
S_i - set of states for process i
o_i - a global initial state
$M_{i,j}$ - a finite set of messages that can be sent from process i to process j
$succ$ - partial function mapping state and message into state.

The model proposed by Lam and Shankar has the following components [Lam and Shankar 84]:

1. A finite set of protocols for the entities $P_1, ..., P_n$
2. A finite set of channels C_i
3. A finite set of messages M_{ik} sent by entity P_i into channel C_k.

The global state in CFSM is specified by a pair $<< s_i >, < m_k >$., where vector $< s_i >$ represents the states of all entities in a system, and m_k represents messages in channel C_k.

A state transition function is replaced here by a set of events. Events are described by two components: an enabling condition and a procedure, which is executed when the enabling predicate is true. Events are classified into two groups: entity events and channel events. The entity events affect both the state of an entity and the contents of the channel. Channel events are only concerned with the messages in a channel. The entity event is defined as a triple:

$$< s_i, s_j, M >$$

where s_i denotes current state, s_j next state, and M denotes sending or receiving a message.

8.4.1 Extended State Transition Language (Estelle)

Extended State Transition Language, *Estelle*, is a formal description technique (FDT) developed by the National Bureau of Standards (NBS). Estelle has been accepted as an international standard by the International Organization for Standardization (ISO) for the description of computer communication protocols [NBS Report 87].

Estelle describes a system as hierarchically organized modules (entities) communicating through channels. A behaviour of an entity is described by a single extended finite state machine. Procedures associated with transitions are described in one of the high level programming languages. Both the hierarchical structure and the use of procedures reduces significantly the *state explosion* problem. Thus, Estelle defines protocol by a relatively small number of states.

Estelle uses the following definition of a transition [Linn 86]:

```
trans
   priority <expression>
   from     <state list>
   to       <state>
   provided <predicate>
   when     <message id>
   delay    <min> <max>
begin
   <procedure description in Pascal>
end;
```

Description used in our model is based on Estelle. However, we do not use any particular programming language for the procedures.

8.5 Representation Methods for FSM

State Transition Diagram

The most often used representation for the FSM model is a *state transition diagram* (state diagram). It is a labeled directed graph, in which arcs represent transitions and nodes represent states. Transitions are labeled by the inputs and outputs. Distinctive states: initial (start), final, and acceptance, are marked by special symbols. This method is a good graphical representation of a machine, but it becomes incomprehensible for a large number of states and transitions. The *state transition table*, the *state transition matrix*, and, especially, *Statechart* accommodate more complex cases.

The State Transition Table and Matrix

The state transition table consists of four columns: current state, input(event), output (action), and next state.

Two representations are used for the state transition matrix. In the first one, the states are listed on the left side of the matrix, events along the top. Each element in the matrix shows an action and next state. In the second representation, the present (current) states are listed on left side and the next states along the top. Events and actions are shown in the elements of the matrix.

Event Trace Diagrams

Event Trace Diagrams show each machine as a vertical line with an assigned name. The messages sent between machines are shown as directed vertical lines from the sender to the receiver. Time changes are shown from top to bottom.

Statecharts

Statecharts, introduced by Harel [Harel 87], extend a finite state machine by four notions: (1) hierarchical (nested) structures of states, (2) state variables, (3) transition actions, and (4) transition guards (Boolean expressions). They are used by many state-based modelling techniques, for example: Object Modelling Technique (OMT) [Rumbaugh et al. 91] and STATEMATE system [Harel et al. 90].

8.6 Communicating Abstract Machines

The formalism described in the following section, *Communicating Abstract Machines* (CAM), combines the features of Abstract Machines, CFSMs, and Statecharts. It is an ***asynchronous*** and ***message-based*** system with composite states. An asynchronous system is understood here as a system which is not synchronized by an internal clock. Transitions, in this model, are enabled only by the messages. However, a synchronous model can be viewed as a special case of the asynchronous approach. Thus, any synchronous model can be represented by the Communicating Abstract Machines, in which the clock signals are described as messages.

In *CAM*, processes (represented by single abstract machines) interact by generating and accepting messages. Messages arrive to the specified machine and are placed on a queue, called a ***mailbox***. A message is removed from the mailbox when the corresponding transition is executed. When the transition is enabled, the machine performs appropriate action and goes to the next specified state.

Composite states are described by an abstract machine, in which the incoming messages for the initial state and the outgoing messages before the final state balance with the incoming and outgoing messages for the decomposed state.

Incoming and Outgoing Messages

There are two general classes of messages: *external* and *internal*. The external messages represent the interaction between processes (machines), and the interaction between the system and its environment. External messages have two types: *incoming* and *outgoing*. The class of internal messages has only one type, called *next-state*. This message is generated by the machine M_i for the machine M_i. Internal messages represent the internal behaviour of the machine and correspond to the λ -transitions in the traditional I/O automata.

Incoming Messages

Incoming messages occur as a result of corresponding outgoing messages. Thus, the message *completeness* means that for each incoming message there must exist a corresponding outgoing message.

Incoming messages play the role of the triggering events for the transitions. An incoming message is represented graphically by a sign ">" and a corresponding message identifier. Thus, "> $m1$" represents incoming message $m1$. Incoming messages have two sources: other machines in the system and the system environment (external control messages: suspend, restart, terminate, or abort).

Outgoing Messages

Outgoing messages are generated by a process (machine) or the system environment. They can be sent to any machine, providing that the receiver's address is known and the mailbox is in operation. To simplify our examples, we assume that the mailboxes are unbounded and always in operation. However, an operating mailbox does not imply an operating receiver.

An outgoing message is represented graphically by a sign "<" and a corresponding message identifier. Thus "< $m1$" represents outgoing message $m1$.

Communication protocols can represent many communication schemas, for example: the *hand-shaking* mechanism or the *posting board* concept. Thus, in the first case, the participant who is sending a message waits until it is acknowledged by the receiver. In the *posting board* scheme, messages are sent to a dedicated participant (with permanent storage) and stored, then they can be read by other participants.

Definition of the asynchronous FSM

Our definition of the asynchronous FSM is influenced by Keller [Keller 76], who defines a transition system as a pair $(Q, \longrightarrow)$, where Q is a set (not necessarily finite set) of states and $\longrightarrow$ is a binary relation $Q \times Q$, called the set of transitions. The asynchronous FSM is defined by the following tuple:

$$FSM = < Q, \Sigma, Q_0, \psi >$$

Where
Q is a state space and $Q \subset S \times V$
Σ is a finite set of incoming and outgoing messages
Q_0 is a set of special states (initial, final, or acceptance), $Q_0 \subset Q$
ψ is a transition function, $\psi : Q \times \Sigma \rightarrow Q \times \Sigma \times \mathcal{F}(V)$

Network of Asynchronous Machines

A collection of FSM machines can be called a network of machines if:

1. There are at least two machines M_i and M_j in the network $Net(\{M_i\}_{i=1}^{n})$
2. For each machine, there exist at least one outgoing and one incoming message.

8.7 Specification Example. Subtask 5

Formal specification of Subtask 5, (case study in Chapter 5) has two parts: (1) Functional Decomposition and (2) Abstract Machine Specification for each participant. The overall communication required for Subtask 5 is illustrated in Figure 25.

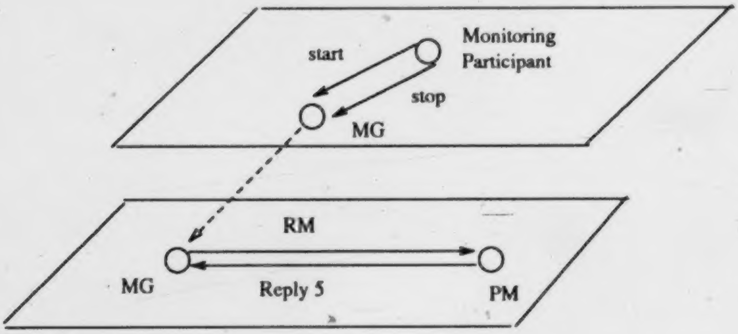

Figure 25: Functional Decomposition of Subtask 5

At Level 1, the Monitoring Participant starts (message ***start***) or stops (message ***stop***) Subtask 5. Level 2 describes communication between the Management Group, *MG*, and the Project Monitoring Group, *PM*. The Management Group sends the message ***Request for Measurements*** (***RM***) requesting current measurements of the customer representatives knowledge. A reply is expected in one week in a form of the Status Report of the Customer Representatives Knowledge, ***Reply 5***.

Figures 26 and 27 represent respectively behaviours of the Management Group and the Project Monitoring Group.

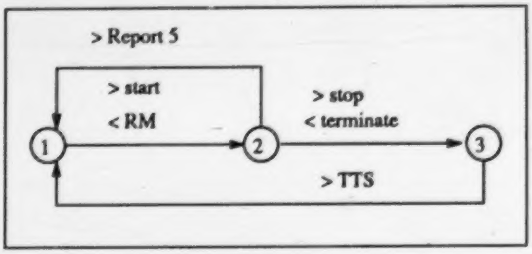

Figure 26: Communication Abstract Machine describing the behaviour of MG

State 1 in Figure 26 is the initial state of the machine. When message *start* is received from the monitoring participant, message *RM* is sent to participant *MG* and the machine changes its state to State 2. In State 2, participant *MG* can receive two messages: *stop* or *Report 5*. When message *stop* comes from the monitoring participant, participant *MG* sends message *terminate* ~~to~~ participant *PM* and goes to State 3. In State 3, it awaits the Terminated Task Status Report, *TTS* from participant *PM*. If, in state 2, the machine receives *Report 5*, it returns to the initial state, State 1, and stops.

Figure 27 describes the behaviour of the Project Monitoring Group (*PM*).

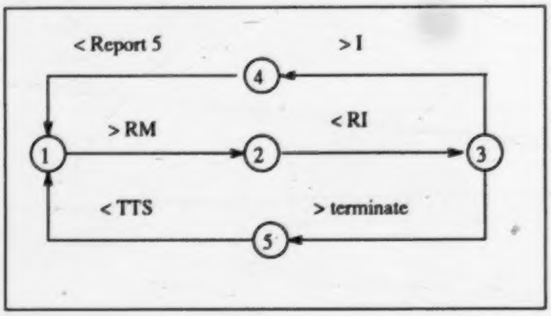

Figure 27: Communicating Abstract Machine describing the behaviour of PM

Initially, the machine is in State 1 (initial state). The incoming message *RM* triggers the first transition — the machine changes its state to State 2. After message *RM* has been received, participant *PM* sends a Request for Information (*RI*) to the Task Librarian (not shown in our example) and goes to State 3. In State 3, it can receive two messages: the requested information (*I*) or *terminate*. When, it receives message *I*, it changes state to State 4, in which the Report 5 is prepared.

When *Report 5* is sent out, the machine changes to State 1 and stops.

When, the machine is in State 3 and message *terminate* arrives, it changes its state to State 5 and prepares message TTS. When message TTS has been sent, the machine returns to its initial state, State 1, and stops.

8.8 Time Representation and its Functions in the Communication Protocol Specification

The *time-sensitive* systems comprise two classes:

1. *Time-dependent* – systems which have *internal* time constraints [Lamport 78]. These constraints are qualitative or quantitative. Qualitative constraints represent the temporal order of events; whereas, the quantitative constraints specify the duration of events.

2. *Real-time* – systems which have *external* time constraints (imposed by their environments). These systems have two types of timing constraints: *performance* and *behavioral* [Dasarathy 82]. Performance constraints set limits on the response time, whereas behavioral constraints describe the rates in which the environment interacts with the system. The *hard* real-time systems have time constraints as a part of their specification and as a condition of correctness. The *soft* real-time systems have time parameters as a part of their specification, but will function correctly even though the time requirements have not been met.

The communication model belongs to the real-time systems. In particular, the protocols for the *critical* tasks are hard real-time. Therefore, the specification and verification of communication protocols is incomplete without time related description and analysis [Yemini 82].

8.8.1 Formal Time Specification

The following subsection provides an overview of time representation in FSM-based models and describes the functions of time in our formalism.

Time in the FSM-Based Model

The FSM-based models use two representations for time; they associate time with (1) the executions of *transitions* or (2) the lifetime of *states*. These two methods correspond to two views of the FSMs. The first one, introduced by Beizer in [Beizer 70], describes states as snapshots of the system at an arbitrary moment of time and transitions as actions of a specific duration. The second approach, described by Bolognesi and Rudin in [Bolognesi and Rudin 84], views transitions as instantaneous.

Time Information

A specific representation for the time constraint is implementation dependent. Thus, depending on particular project and implementation, time can be *global* or *local*, and it can be represented by *constants* or *intervals* ([Bolognesi and Rudin 84], [Dasarathy 82]). Furthermore, the notion of time can be used in various functions. We will discuss here three functions of time: (1) reliable communication, (2) performance specification, and (3) synchronization with external systems.

Mechanism Providing a Reliable Communication

A mechanism to improve the reliability of communication is needed, simply because the human participants are unreliable. Thus, we present in this section an example which uses time-outs, acknowledgements, and reminders.

The time-outs are supported by a mechanism denoting the passage of fixed amounts of time (local or global).

An acknowledgement mechanism requires that the receiver sends back an acknowledgement upon receiving the message. If the acknowledgement is not received by the sender, after a specified time, a reminder is sent.

A simple model with acknowledgements and reminders is described by the following scenario. Participant S sends message m to participant R. When message m is received by R, R sends back an acknowledgement $ACK(m)$ for message m. When acknowledgement $ACK(m)$ is received by S, predicate $RECEIVED(ACK(m))$ is evaluated to true.

The process of resending reminder *rem* is described by the following procedure:

```
Send (m);
While not RECEIVED(ACK(m)) do
     Send (rem)
end;
```

This procedure will resend a reminder until the acknowledgment is received. However, if the receiver is not operating, the sender sends unlimited number of unnecessary reminders. Therefore , this simple mechanism should be modified to restrict the number of reminders. This can be achieved by two mechanisms: a *time-out* and a *counter* of the reminders. First mechanism, a *time-out*, resends a reminder at specific time intervals. The time-out value can be approximated from the message propagation time, priority of the message, and the receiver workload (receiver response time). The second mechanism limits the number of reminders; after the maximum value is reached, the receiver is assumed to be non-active.

The timer is set up for each message, m, and it can be disabled only when the predicate $RECEIVE(ACK(m))$ is true. A reminder *rem* is sent when TIMEOUT(m) from the timer is received.

```
Send (m);
Set timer for interval i;
While not RECEIVED(ACK(m))  and TIMEOUT(m) do
    set timer for interval i
    send (rem)
end;
```

In the mechanism with a counter, reminders are resent a limited number of times. An arbitrary number n is chosen to represent a maximum number of retries. If after n reminders the acknowledgment is not received, this procedure stops resending and initiates a task to inform about a communication problem.

```
COUNT:=0;
While not RECEIVED(ACK(m)) and TIMEOUT(m) and COUNT < n do
    set timer for interval j
    send (rem)
    COUNT:= COUNT + 1
end;
If COUNT > = n
   Start Communication Problem Task;
```

Mechanism for the Performance Specification Measurements

Performance constraints require estimation of the execution time for tasks. The time estimation is done by the participants based on the history of similar tasks and the available project resources. For example, in Subtask 5, the Management Group estimated the time required for Report 5 for one week.

Mechanism for Synchronization with External Systems

The real time mechanism is required, when two or more systems must be synchronized at some point. Real time provides the external reference point to all participating systems. Time requirements can be generalized as a synchronization between real time and a particular state of a system. This synchronization can be represented as a triple (s, t_{max}, t_{min}). When the minimum time is specified, the system must be in state s at the time t_{min} or after time t_{min}. When the maximum time is specified, the system must be in state s before or at the time t_{max}. Combination of these two will give a specification: the system must be in state s after time t_{min} and before time t_{max} .

8.8.2 Example. Timing Specification for Subtask 5

Figure 28 illustrates an abstract machine specifying Subtask 5. It corresponds to the behaviour of Management Group depicted in Figure 8.4.

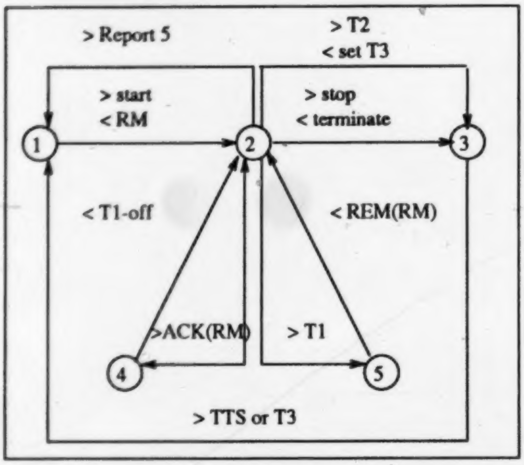

Figure 28: A Communicating Abstract Machine with Time-outs

The example above has three time-outs: (1) $T1$ for message RM, (2) $T2$ for *Report 5*, and (3) $T3$ for TTS. The time-outs set the timer to appropriate values. After the specified time t_n, a time-out message T_n is sent by the timer (time server). Time-outs T_1, T_2, and T_3 have two functions. Time-out $T1$ is used to improve the reliability of communication, while $T2$ and $T3$ are used as performance constraints. T_2 is the estimated time for the Report 5 and T_3 is the time required to complete the Terminated Task Status Report (TTS).

Initially, the machine is in State 1. When message *start* is received, it sends out message RM and goes to State 2. Message RM automatically sets timers $T1$ and $T2$ on. Timer $T1$ is set to , for example, one hour; timer $T2$ is set to one week. Thus, if an acknowledgement for message RM ($ACK(RM)$) has not been received after $t1$ units of time, a time-out message $T1$ is sent by the timing mechanism. When $T1$ is received, the machine goes to State 5, sends out a reminder ($REM(RM)$), and returns to State 2.

However, if the acknowledgment $ACK(RM)$ is received, the machine changes to State 4, turns off timer $T1$ by sending message *T1-off*, and returns to State 2.

If the *Report 5* has not been received before the timeout $t2$, the machine sets timer $t3$ and goes to State 3. When, the time-out $T3$ or message TTS is received, the machine goes to the final state (State 1).

Formal Specification of Subtask 5

```
Management Group Subtask 5, Level 2

Participant:       Management Group (MG)
States     :       S1, S2, S3
Initial State:     S1
Finial State :     S1
Desired Outcome:   Report 5

Transition from State 1 to State 2
           when message:  start
           action:        send message RM
                          set timer T1 and T2

Transition from State 2 to State 3
           when message:  time-out T2
           action:        set timer T3
                          send message request TTS

Transition from State 2 to State 3
           when message:  stop
           action:        set timer T3
                          send message terminate

Transition from State 2 to State 1
           when message:  Report 5
           action:        next state

Transition from State 2 to State 4
           when message:  ACK(RM)
           action:        next state

Transition from State 4 to State 2
           when message:  next state
           action:        T1-off
```

```
Transition from State 2 to State 5
           when message:  timeout T1
           action:        next-state

Transition from State 5 to State 2
           when message:  next-state
           action:        REM(RM)

Transition from State 3 to State 1
           when message:  TTS or timeout T3
           action:        next-state
```

Chapter 9

Conclusions

9.1 Contribution of this thesis

This thesis describes a new model for the software systems development process – a model based on technical communication between all project participants. The contributions of the communication model to software systems development process modelling, can be summarized by the following:

1. The communication model recognizes the most important aspect of the software systems development process – communication. It formally models the communication processes between all project participants (Chapter 1 discusses the role of communication).

2. The communication model emphasizes the role of people. It includes explicit representation of all project participants, not only developers. An explicit representation of participants and a formal description of the communication processes can be used to specify a software system in terms of *services*.

   Human aspects were recognized early in the history of software engineering, particularly by the well known book *The Mythical Man-Month* [Brooks 75]. However, research on the role of people in project development have concentrated on human resources management [Abdel-Hamid 89], computer psychology, and cognitive issues. The human dimension has not gained attention in the formal models for the software systems development process.

3. The communication model uses a uniform paradigm for the entire life of the software system – it does not distinguish between so-called "initial development" and "maintenance".

4. The communication model considers both types of communication: oral and written. Therefore, it unifies product-oriented (document-driven and code-

driven methods) and process-oriented (risk-driven and process programming) approaches. The mapping between various models (waterfall, spiral, and rapid prototyping) and the communication model is described in Section 4.5.

5. The communication model is formally defined using operational semantics; therefore, it can be executed (enacted) by people and machines.
6. The communication model is a *generic* model; therefore, it can be instantiated in specific projects using different methods and techniques.
7. *Communication* is a generic type of activity that can describe all types of development activities: negotiation, learning, and managing. Furthermore, it is used as a common description for software systems and their development process.
8. The communication model can improve many aspects of project management; it eliminates unnecessary communication, facilitates communication between participants distributed among remote sites, improves the task understanding, and provides automated task documentation (Section 2.8.2 and Section 3.4.1 describe this issue in more detail).
9. The formal specification of communication provides methods to identify specific properties of the system, such as: absence of deadlock, presence of bottlenecks, and completeness (Section 3.4.2 provides more examples).

9.2 Future Work

Our communication model integrates the CASE (Computer Aided Software Engineering) tools with systems development methodologies. In this model, the methods and techniques are described by the communication protocols and behavioral constraints. The sets of processes and constraints are implementation and project specific. The users of each software environment can define their own sets of processes, for example, using an Environment Definition Language (EDL) [Sorenson 88a] [Sorenson 88b].

The communication model can be used to integrate the tools for software engineering environments. This system can monitor the entire life of tasks and subtasks of the project. It can give information about the history and current status of all components: participants, tasks, modules, knowledge repositories, and resources.

The system based on the communication model can support the following facilities:

1. Documentation Mechanism for the project tasks, participants, modules, knowledge repositories, and resources (multi-version mechanism, history mechanism).
2. Reporting Facility for the current status of the system in terms of tasks, participants, associated modules, and assigned resources.
3. The Task Planning, Task Scheduling and Rescheduling Facility (producing the PERT charts, Gantt charts, and automated analysis of the critical paths)
4. Automated Analysis of communication bottlenecks, deadlocks, and lifelocks.
5. Rule-based Task Assigning Mechanism using a matching and reasoning process.
6. Analysis of the work (task) load for individual participants
7. A Security Mechanism for project modules and project knowledge repositories
8. Concurrency Control Mechanism for project modules

Most of the information required by this model is already present in any large project. However, related data is scattered among the organizational charts and standards, reporting rules, time sheets, progress reports, and task assignments. The proposed system gathers the existing information in a well-organized communication network.

Appendix A

Implementation Examples for the Communication Model

A.1 Participant Implementation

In the example below, the participant attributes are organized into three sets: *control*, *expertise*, and *security*. The control attributes describe the level of authority. The highest level allows for the task initialization, execution, modification, and termination. The expertise area describes the knowledge related to a particular repository (application, software, hardware, methodology). The level of knowledge is described by a standardized scale (for example: years of experience). The expertise is described as required or desired. Required expertise is necessary to carry the participant functions; whereas, desired expertise is helpful but not necessary. The security level defines the access rights to the sensitive modules and knowledge repositories.

An Example of a Participant Specification

Participant *Task Initiator* is described by a set of attributes:

```
Participant Identifier:   task initiator 1
    Attributes:
        Control:  initiate and terminate tasks for system support
        Expertise:
            Required:
                 Application knowledge: production system, Level=2
                 Software knowledge:    SQL/DS, Level=1,
                                        COBOL, Level=2
                 Methodology:           Waterfall model
    Desired:
         Software knowledge:     SQL/DS with REXX, Level=1
Security level:
      Exclude: control task
```

A.1.1 Resources Assigning Procedure

The project staffing is represented by a mapping $\mathcal{F} : P_e \rightarrow \mathcal{P}(R_e)$ between project participants and the project resources: human, software, and hardware. A set of project resources is assigned to each participant. When there are no resources to assign, the set is empty. The mapping $\mathcal{F}$ can be dynamically changed, allowing for an easy reassignment of participants. This flexibility is particularly important for the unusually high staff turnover in the information system projects.

A.1.2 Human Resources Assigning Procedure

Human resources have a special place in the communication model. The process of assigning tasks to resources (staffing) requires analysis of two goals: the project goal and the career and life goal of the people participating in the project. Therefore, the matching process is based on a number of rules considering all aspects of human resource management [Boehm 81].

In a particular implementation, the Task Assigning process can involve two types of procedures: automated and manual.

Automated Assigning Procedure

The initial mapping $\mathcal{I} : P_e \rightarrow \mathcal{P}(R_e)$ between participants and project resources matches the lists of attributes and the workload of a particular person with the specific participant instance. Each participant is mapped into a set of possible candidates from human resources. The set of candidates is ordered by a *weight*. Weight reflects the degree of similarities between participant and resource attributes. Participants have two classes of qualifications: required and desired. Required qualifications are necessary to include a human resource member in a matching set. Desired qualifications are used to calculate the *weight* of each resource. The human resource matching process is described by the user defined rules. The following two rules can be used, for examples, to calculate the weight value for the candidates.

```
Matching Rule 1:

  If resource.control  matches  participant.control and
     resource.security matches  participant.security and
```

```
    (for each type of required expertise
        resource.level >= participant.level) then
    do
       add resource-id to candidate-list
       resource-id.weight := resource.level - participant.level
    end
```

```
Matching Rule 2:

  If resource-id in candidate-list then
    do
       for each type of desired expertise
       resource-id.weight := resource-id.weight +
                             resource.level X 2
    end
```

This initial automatic mapping gives a suggested list of candidates, which can be easily modified by the authorized participants to reflect the specific situation and individual preferences of people.

The reassignment procedure takes specified human resources and finds the corresponding participants. Subsequently, it performs the initial assignment procedure for all affected participants.

Manual Procedure for Task Assignment

The manual procedure allows for the selection from the automatically created list of candidates. Moreover, it permits ad hoc changes to project staffing.

A.1.3 Human Resources Assigning Example

The initial participant assignment procedure produces a set of project resources for the participant instance *task-initiator-1*. The set of possible candidates has two elements: employee id 900808 and employee id 870701. Candidate 900808 has a weight of 10, and candidate 870701 has a weight of 9.

```
Set of candidates = {(900808, 10), (870701, 9)}
```

The weight on the candidate list is calculated according to prespecified rules.

```
Matching RULE 1:

Remark:
  p.control  corresponds to participant's control attribute
  r.control corresponds to resource's control attribute

If p.control = r.control and
   p.security = r.security  and
   p.required expertise level < or = r. expertise level
   then
      add resource identifier to the candidate list
      weight = 1 + (r. expertise level - p.required-level) X 2
               + (p.expertise level - r.desired expertise level)
```

The first person assigned to the system support task has the following description:

```
Resource Identifier:  900808
Resource Type:        Human
Attributes:
    Control:                      system support
    Expertise:
       Application knowledge: production system
       Software knowledge:    SQL/DS, Level=3
                              SQL/DS with REXX, Level=2.5
       Methodology:           Waterfall model
    Security level:           No exclusion
```

The second person has the following description:

```
Resource Identifier:  870701
Resource Type:  Human
Attributes:
    Control:                      system support
    Expertise:
       Application knowledge: production system
       Software knowledge:    SQL/DS, Level=2
       Methodology:           Waterfall model
    Security level:           No exclusion
```

A.2 Services, Mechanisms, and Facilities

Services, mechanisms, and facilities constitute a part of model *implementation.* They support the software systems development process and automate execution of development activities. However, their implementation details are not crucial to the conceptual understanding of the communication model. Yet, we will include a few implementation examples to demonstrate the universality of the communication model.

A particular implementation of the communication model can have services for each component: participants, tasks, modules, resources, and knowledge repositories. Services can use common mechanisms, such as history, concurrency control, and version control. At the conceptual level, services, mechanisms, and facilities can be viewed as high level participants providing specialized functions.

A.2.1 Services

The following subsection describes four object related services: Module, Resource, Knowledge Repository, and Resource; and three task related services: Defining and Planning, Scheduling, and Execution.

The Project Modules Service manages the Project Modules. It has two supporting mechanisms: a multi-version mechanism and a concurrency control mechanism.

The Resource Service is responsible for maintaining the current and historical data about resources (human, hardware, software, financial), as well resource allocation. It has two mechanisms: allocation and history.

The Knowledge Repositories Service maintains a large knowledge base for applications, methodologies, techniques, and standards. It also provides advisory and tutoring services. The advisory service can provide information about methodologies, techniques, standards, and task examples.

The Task Defining and Planning Service is used to specify tasks and their participants. The resource allocation is done by the same service using the allocation mechanism from the Resource Service. The Task Scheduling Service provides the scheduling and rescheduling of tasks. It also verifies resource allocation. Task scheduling can also be equipped with a history mechanism as an auxiliary. The Task Execution Service starts, controls, and terminates task execution. It also has access to the history of previous executions.

Figure 29 illustrates the organization of Services and Mechanisms for the described implementation of the communication model.

A.2.2 The Advisory Mechanism Example

The Advisory Mechanism provides information to an inexperienced participant. For example, a new participant is given a task involving a Data Flow Diagram for a small component of the system. When the participant issues a request, such as: "Provide all information about DFD"; the Advisory Mechanism will provide the following information:

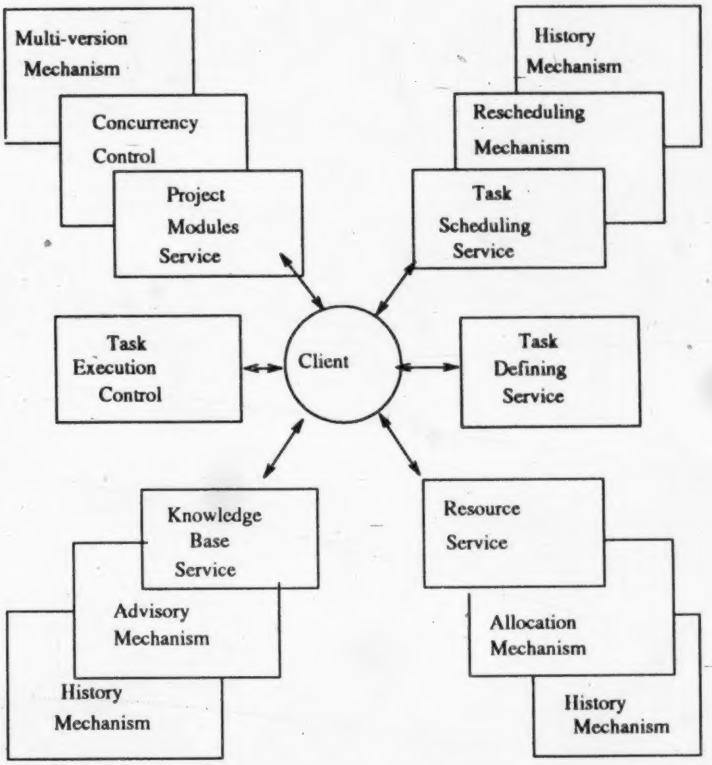
Multi-version Mechanism
Concurrency Control
Project Modules Service
History Mechanism
Rescheduling Mechanism
Task Scheduling Service
Task Execution Control
Client
Task Defining Service
Knowledge Base Service
Advisory Mechanism
History Mechanism
Resource Service
Allocation Mechanism
History Mechanism

Figure 29: Structure of Services and Mechanisms

Information related to Data Flow Diagrams:

1. Standard notation used in this project: Gane-Sarson notation
2. Identification Standards (process - Pn, each lower level Pn.n)
3. Tools: Exelerator
4. Restriction: all objects (process, data flow, external/internal entity, data store) must be documented in the project dictionary
5. DFD examples: list of existing DFD
6. DFD designing task example: Task specification
7. Example of task execution

A.2.3 Reporting Facilities

Implementation of the communication model can provide historical, statistical, and current status reports. These reports can be requested on a regular (weekly, monthly) or an ad hoc basis. The appropriate reports will be produced in the specified time period and distributed (for example: by electronic mail) to the users.

A similar reporting facility, providing selected news, was described by Gifford [Gifford 90]. This system has an active agent searching for information on a daily basis and providing only selected information (from newspapers, journals, etc.) to its customers. Similarly, in the implementation of our model, the user can specify an interest topic or a report type; and an appropriate participant will search periodically the literature and prepare reports.

A rigorous description of the project participants and the development activities enable automated reporting. Thus, the system can produce various reports. For example: (1) Project Tasks Reports (current status of tasks presented in descriptive or graphical form), (2) Resource Allocation Reports – showing the resources with assigned tasks, (3) Resource Utilization Reports – showing resource schedules.

The historical mechanisms maintain information about the history of all project components. Thus, historical reports can also be automatically generated by the system.

Bibliography

[Abdel-Hamid 89] Tarek K. Abdel-Hamid, The Dynamics of Software Project Staffing: A System Dynamics Based Simulation Approach, *IEEE Trans. on Software Engineering*, Vol. 15, No. 2, (February 1989), pp. 109-119.

[Amadio 89] William Amadio, *Systems Development. A Practical Approach*, Michell Publishing, Inc., 1989.

[Amer et al. 88] Paul D. Amer, Figen Ceceli, and Guy Juanole, Formal Specification of ISO Virtual Terminal in Estelle, *IEEE INFOCOM'88*, March 1988.

[Andrews 91] Dorine C. Andrews, JAD: A Crucial Dimension for Rapit Application Development, *Journal of Systems Management*, March 1991, pp. 23-27.

[Ansart 86] J.P. Ansart et al, Software Tools for ESTELLE, *Protocol Specification, Testing, and Verification, VI*, 1987, pp.55-61.

[Agha 86] Gul Agha, *Actors. A Model of Concurrent Computation in Distributed Systems*, The MIT Press, 1986.

[Aho et al. 79] . V. Aho, J.D. Ullman, and M. Yannakakis, Modelling Communications Protocols by Automata, *Proceedings of the 20th Symposium on Foundations of Computer Science*, October 1979, San Juan, Puerto Rico, pp. 267-273.

[Armenise 89] Pasquale Armenise, A structured approach to program optimization, *IEEE Trans. on Software Engineering*, Vol. 15, No. 2, (February 1989), pp. 101-108.

[Ashok et al. 89] V. Ashok, J. Ramanathan, S. Sarkar, and V. Venugopal, Process Modeling in Software Engineering, *Software Engineering Notes*, Vol. 14, No. 4, pp. 39-42.

[Avizienis 85] A. Avizienis, The N-version approach to fault-tolerant software, *IEEE Trans. on Software Engineering*, Vol. 11, No. 12, (December 1985), pp. 1491-1501.

[Balzer and Cheatham 84] Robert Balzer and Thomas E. Cheatham, Software Technology in the 1990's: Using a New Paradigm, *Proceedings of the Software Process Workshop*, Surrey, UK, February 1984, pp. 3-11.

[Balzer 86] Robert Balzer, Program Enhancement, *ACM SIGSOFT Software Engineering Notes*, Vol. 11, No. 4, August 1986, pp. 66-67.

[Beizer 70] B. Beizer, Analytical Techniques for the Statistical Evaluation of Program Running Time, *Proc. Fall Joint Computer Conference*, 1970, pp. 519-524.

[Belady and Lehman 76] L. A. Belady and M. M. Lehman, A Model of large program development, *IBM Systems Journal*, Vol. 15, No. 3, 1976, pp. 225-252.

[Blyth et al. 90] David Blyth, Cornelia Boldyreff, Clive Ruggles, and Nik Tetteh-Lartey, The Case for Formal Methods in Standards, *IEEE Software*, September 1990, pp.65-67.

[Blumer 82] Thomas P. Blumer and Richard L. Tenney, A Formal Specification Technique and Implementation Method for Protocols, *Computer Networks*, 6 (1982), pp.201-217.

[Bochmann and Gecsei 77] Gregor V. Bochmann and Jan Gecsei, A Unified Method for the Specification and Verification of Protocols, *IFIP*, 1977, pp.229-234.

[Bochmann 78] Gregor V. Bochmann, Finite State Description of Communication Protocols, *Computer Networks* , 2 (1978), pp.361-372.

[Bochmann and Sunshine 80] Gregor V. Bochmann and Carl A. Sunshine, Formal Methods in Communication Protocol Design, *IEEE Trans. on Communications*, Vol. COM-28, No.4, April 1980,

[Bochmann 80] Gregor V. Bochmann, A General Transition Model for Protocols and Communication Services, *IEEE Trans. on Communications*, Vol.COM-28, No.4, April 1980, pp.643-650.

[Bochmann et al. 82] Gregor V. Bochmann et al., Experience with Formal Specifications Using an Extended State Transition Model, *IEEE Trans. on Communications*, Vol.COM-30, No.12, December 1982, pp.2506-2511.

[Bochmann 87] Gregor V. Bochmann, Semiautomatic Implementation of Communication Protocols, *IEEE Trans. on Software Engineering*, Vol. SE-13, No.9, September 1987, pp.989-999.

[Boehm 81] Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[Boehm 87] Barry W. Boehm, Improving Software Productivity, *IEEE Computer*, September 1987, pp. 43-57.

[Boehm 88] Barry W. Boehm, A Spiral Model of Software Development and Enhancement, *IEEE Computer*, May 1988, pp. 61-72.

[Boehm and Ross 89] B. W. Boehm and R. Ross, Theory-W Software Project Management: Principles and Examples, *IEEE Trans. on Software Engineering*, Vol. 15, No. 7 (July 1989), pp. 902-915.

[Bolognesi and Rudin 84] T. Bolognesi and H. Rudin, On the analysis of time-constrained protocols by network flow algorithms, *Proc. Workshop on Protocol Specification, Testing, and Verification, IV*, Sky Top, Pennnsylvania, June 1984, (North-Holland, Amsterdam, 1984).

[Borenstein 91] Nathaniel S. Borenstein, Multimedia Electronic Mail: Will the Dream Become a Reality?, *CACM*, Vol. 34, No. 4, (April 1991), pp. 117-119.

[Borenstein and Thyberg 91] Nathaniel S. Borenstein and Chris A. Thyberg, Power, ease of use and cooperative work in a practical multimedia message system, *International Journal of Man-Machine Studies*, Vol. 34, No. 3, February 1991, pp. 229-259.

[Borgida et al. 87] Alex Borgida, Mathias Jarke, John Mylopoulos, Joahim W. Schmidt and Yannis Vassiliou, A knowledge Based Environment for Building Information Systems, *Proceedings of CIPS Conference*, Edmonton, 1987, pp. 99-107.

[Bostrom 84] Robert P. Bostrom, Development of Computer-Based Information Systems – A Communication Perspective, *Computer Personnel*, Vol. 9, No. 4, (August 1984), pp. 17-25).

[Bostrom 88] Robert P. Bostrom, A New Memeber Of Your Management Team, *Information Executive*, Vol. 1, No. 1, pp. 43-46.

[Bostrom 89] Robert P. Bostrom, Successful Application of Communication Techniques to Improve the Systems Development Process, *Information and Management*, Vol. 16, No. 5, (May 1989), North-Holland, pp. 279-295.

[Bowles 90] Adrion J. Bowles, A Note on the Yourdon Structured Method, *Software Engineering Notes*, Vol. 15, No. 2, April 1990, p.27.

[Brand and Zafiropulo 83] Daniel Brand and Pitro Zafiropulo, On Communicating Finite-State Machines, *Journal of the ACM*, Vol. 30, No. 2, (April 1983), pp. 323-342.

[Brooks 75] Frederick P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley, 1975.

[Budkowski and Dembinski 87] Stanislaw Budkowski and Piotr Dembinski, An Introduction to Estelle: A Specification Language for Distributed Systems, *Computer Networks and ISDN Systems* 14 (1987), pp. 3-23.

[Burns and Dennis 85] R. N. Burns and A. R. Dennis, Selecting Application Development Methodology, *Data Base*, Vol. 17, No. 1, Fall 1985, pp. 19-23.

[Chan and Henderson-Sellers 90] M. L. Chan and B. Henderson-Sellers, Corporate Object-oriented Development Environment (CODE), *ACM SIFSOFT*, Vol. 15, No. 1, January 1990, pp. 42-43.

[Choi 84] Tat Y. Choi, On the recoverability of finite state protocols, *Proc. of the COMPSAC '84 Conference*, pp.325-332.

[Choi 85] Tat Y. Choi, Formal Techniques for the Specification, Verification and Construction of Communication Protocols, *IEEE Communication Magazine*, Vol.23, No.10, October 1985, pp. 46-52.

[Clark 90] Jon D. Clark, Function Versus Data-Driven Methodologies: A Prescriptive Metric, *Software Engineering Notes*, Vol. 15, No. 2, April 1990, p. 26.

[Constantine 90] L. Constantine, Teamwork Paradigm and the Structured Open Team, *Proceedings of Software Development '90*, San Francisco, 1990.

[Conway 68] Melvin B. Conway, How Do Committees Invent?, *Datamation*, April 1968, pp. 28-31.

[Cook et al. 91] Steve Cook, Gary Birch, Alan Murphy and John Woolsey, Modelling groupware in the electronic office, *International Journal of Man-Machine Studies*, Vol. 34, No. 3, March 1991, pp. 369-393.

[CSTB Report 90] Computer Science Technical Board, *CACM*, Vol. 33, No. 3, (March 1990), pp. 281-293.

[Curtis et al. 87] Bill Curtis, Herb Krasner, Vincent Shen and Neil Iscoe, On Building Software Process Models Under the Lamppost, *Proceedings of the 9th International Conference on Software Engineering*, Monterey, California, March 1987, pp. 96-103.

[Curtis et al. 88] Bill Curtis, Herb Krasner, and Neil Iscoe, A Field Study of the Software Process for Large Systems, *CACM*, Vol. 31, No. 11 (November 1988), pp. 1268-1287.

[Curitis 89] Bill Curtis, Three Problems Overcome with Behavioral Models of the Software Development Process, *Proceedings of the 11th International Conference on Software Engineering*, May 1989, pp. 398-399.

[Daly 79] Edmund B. Daly, Organizing for Successful Software Development, *Datamation*, December 1979, pp. 107-120.

[Davis et al. 88] Alan M. Davis, Edward H. Bersoff, and Edward R. Comer, A Strategy for Comparing Alternative Software Development Life Cycle Models, *IEEE Trans. on Software Engineering*, Vol. 14, No. 10, October 1988, pp. 1453-1460.

[Danthine 80] Andre A. S. Danthine, Protocol representation with finite-state models, *IEEE Trans. on Communications*, vol. 28, no. 4, pp. 632-643, 1980.

[Dasarathy 82] J. Dasarathy, Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them, *Proc. IEEE Real-Time Systems Symposium*, December 1982.

[Deiters et al. 89] Wolfgang Deiters, Volker Gruhn, and Wilhem Schafer, Systematic Development of Formal Software Process Models, *Proceedings of 2nd European Software Engineering Conference*, Coventry, UK, September 1989, LNCS 387, pp. 100-117.

[Dembinski and Budkowski 87] Piotr Dembinski and Stanislaw Budkowski, Simulation Estelle Specifications with Time Parameters, *Proc. IFIP Seventh International Conference on Protocol Specification, Testing and Verification*, Montreal, June 1986.

[Diaz 82] Michael Diaz, Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models, *Proc. 2nd Int. Workshop on Protocol Specification, Testing and Verification*, May 1982.

[Diaz 87] Michel Diaz, Petri Net Based Models in the Specification and Verification of Protocols, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, Lecture Notes in Computer Science, Vol. 255, Springer-Verlag 1987.

[Dixon 88] David Dixon, Integrated Support for Project Management, *Proceedings of the 10th International Conference on Software Engineering*, Singapore, 1988, pp. 49-58.

[Dowson 86] Mark Dowson, The Structure of the Software Process, *ACM SIGSOFT Software Engineering Notes*, Vol. 11, No. 4, August 1986, pp. 6-10.

[Ellis et al. 91] C. A. Ellis, S. J. Gibbs, and G. L. Rein, Groupware: Some Issues and Experiences, *CACM*, Vol. 34, No. 1 (January 1991), pp. 38-58.

[Fernstrom 89] Christer Fernstrom, Design Considerations for Process Driven Software Environments, *Software Engineering Notes.* Vol. 14, No. 4, June 1989, pp.65-67.

[Finkelstein and Fuks 89] Anthony Finkelstein and Hugo Fuks, Multi-party Specification, *ACM SIGSOFT Software Engineering Notes*, Vol. 14, No. 3, (May 1989), pp. 185-195.

[Finkelstein et al. 89] Anthony Finkelstein, Hugo Fuks, Celso Niskier and Martin Sadler, Constructing a Dialog Framework for Software Development, *Software Engineering Notes.* Vol. 14, No. 4, June 1989, pp.68-72.

[Finegan an Besnier 89] Edward Finegan and Niko Besnier, *Language Its Structure and Use*, HBJ Publishers, 1989.

[Flaatten et al. 89] Per O. Flaatten, Donald J. McCubbrey, P. Declan O'Riordan, Keith Burgess, *Foundations of Business Systems*, The Dryden Press, 1989.

[Floyd 89] C. Floyd, STEPS to Software Development with Users, *ESEC'89 2nd European Software Engineering Conference*, LNCS 387, Springer Verlag, 1989, pp. 48-64.

[Gifford and Francomano 90] David K. Gifford and Anne Francomano, An Information System Based Upon Programmable Agents, *Proceedings of the CIPS Conference*, Edmonton, 1990, Session 4, Paper 1.

[Gladden 82] G. R. Gladden, Stop the Life-Cycle, I Want to Get off, *Software Engineering Notes*, Vol. 7, No.2, April 1982, pp. 35-39.

[Gomaa 83] Hassan Gomaa, The Impact of Rapid Prototyping on Specifying User Requirements, *ACM SIGSOFT Software Engineering Notes*, Vol. 8, No. 2, April 1983, pp. 17-28.

[Goldberg 90] Allen Goldberg, Reusing Software Developments, *ACM SIGSOFT Software Engineering Notes*, Vol. 15, No. 6, (December 1990), pp. 107-119.

[Greenberg 91] Saul Greenberg, Computer-Supported Cooperative Work and groupwork, *International Journal of Man-Machine Studies*, Vol. 34, No. 2, February 1991, pp. 133-142.

[Grohowski et al. 90] Ron Grohowski, Chris McGoff, Doug Vogel, Ben Martz, Jay Nunamaker, Implementing Electronic Meeting Systems at IBM: Lessons Learned and Success Factors, *MIS Quarterly*, Vol. 14, No.4, December 1990, pp. 369-383.

[Guinan and Bostrom 86] Patricia J. Guinan and Robert P. Bostrom, Development of Computer-Based Information Systems: A Communication Framework, *Data Base*, Vol. 17, No.3, Spring 1986, pp.3-16.

[Guttag 81] J. Guttag, A Few Remarks on Putting Formal Specification to Productive Use, *Proceedings of the Workshop on Program Specification* , Aarthus, Denmark, August 1981, LNCS 134, Springer Verlag, 1981, pp. 370-380.

[Hall 90] Anthony Hall, Seven Myths of Formal Methods, *IEEE Software*, September 1990, pp. 11-19.

[Harel 87] David Harel, Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, Vol. 8, No. 3, (June 1987), pp. 231-274.

[Harel 88] David Harel, On Visual Formalisms, *CACM*, Vol. 31, No. 5, (May 1988), pp. 514-530.

[Harel et al. 90] David Harel, Hagi Lachover, Ammon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot, STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Trans. on Software Engineering*, Vol. 16, No. 4, (April 1990), pp. 403-413).

[Harston and Hix 90] H. Rex Harston and Deborah Hix, Developing Human-Computer Interface Models and Representaiton Techniques, *Software Practice and Experience*, Vol. 20, No. 5, (May 1990), pp. 425-457.

[Hatley and Pirbhai 87] Derek J. Hatley and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing, 1987.

[He and Lee 88] Xudong He and J. A. N. Lee, A Strategy for Integrating Formalisms in Software Development, *Proceedings of CIPS'88 Conference*, Edmonton 1988, pp. 33-42.

[Henderson-Sellers and Edwards 90] Brian Henderson-Sellers and Julian M. Edwards, The Object-Oriented Systems Life Cycle, *CACM*, Vol. 33, No. 9 (September 1990), pp. 143-159.

[Hirschheim and Klein 89] Rudy Hirschheim and K. Klein, Four Paradigms of Information Systems Development, *CACM*, Vol. 32, No. 10, (October 1989), pp. 1199-1216.

[Hoare 85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[Horning 81] J. J. Horning, Program Specification: Issues and Observations, *Program Specification, Proceedings of a Workshop , Aarthus, Denmark, August 1981*, LNCS 134, Springer Verlag, 1981, pp. 5-24.

[Humphrey 89] Watts S. Humphrey, The Software Engineering Process: Definition and Scope, *Software Engineering Notes*, Vol. 14, No. 4, (June 1989), pp. 82-83.

[Jackson 83] Michael Jackson, *System Development*, Prentice-Hall, 1983.

[Jain 87] Pradeep Jain and Simon S. Lam, Modeling and Verification of Real-Time Protocols for Broadcast Networks, *IEEE Trans. on Software Engineering*, Vol. SE-13, No.8, August 1987, pp.924-937.

[Jard 85] Claude Jard, Roland Groz, Jean-Francois Monin, VEDA: a software simulator for the validation of protocol specification, *Proc. COMNET'85*, Budapest, North-Holland 1985.

[Johnson-Lenz 91] Peter Johnson-Lenz and Trudy Johnson-Lenz, Post-mechanistic groupware primitives: rhythms, boundaries, and containers, *International Journal of Man-Machine Studies*, Vol. 34, No. 3, March 1991, pp. 391-417.

[Josephs 88] Mark B. Josephs, A state-based approach to communicating processes, *Distributed Commputing*, Vol. 3, No. 1, 1988, pp. 9-18.

[Katayama 89] Takuya Katayama, A Hierachical and Functional Software Process Description and its Enaction, *Proceedings of the 11th International Conference on Software Engineering*, Nice, France, 1989, pp. 343-352.

[Kelier 76] Robert M. Keller, Formal Verification of Parallel Programs, *CACM*, Vol. 19, No. 7, July 1976.

[Kellner 89] Marc I. Kellner, Representation Formalisms for Software Process Modeling, *ACM SIGSOFT Software Engineering Notes*, Vol. 34, No. 4, (June 1989), pp. 93-96.

[Kettelhut 91] Michael C. Kettlelhut, Avoiding Group-Induced Errors in Systems Development, *Journal of Systems Management*, March 1991, pp. 13-20.

[Koubek et al. 89] Richard J. Koubek, Gavriel Salvendy, Hubert E. Dunsmore and William K. LeBold, Cognitive issues in the process of software development: review and reappriasal, *International Journal of Man-Machine Studies*, Vol. 30, (1989), p. 171-191.

[Kovacs 86] Laszlo Kovacs and Andras Ercsenyi, Specification Versus Implementation Based on ESTELLE, *Computer Communication Review*, Vol. 16, No. 4, July/August 1986.

[Kozar 88] Kenneth A. Kozar, *Humanized Information Systems Analysis and Design*, McGraw-Hill, 1988.

[Lam and Shankar 84] Simon S. Lam and A. Udaya Shankar, Protocol Verification via Projections, *IEEE Trans. on Software Engineering*, Vol. SE-10, No.4, July 1984, pp.325-342.

[Lam and Shankar 90] Simon S. Lam and A. Udaya Shankar, A Relational Notation for State Transition Systems, *IEEE Transactiona on Software Engineering*, Vol. 16, No. 7, (July 1990), pp. 755-775.

[Lamport 78] Leslie Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *CACM*, Vol.21, No.7, July 1978, pp.558-565.

[Lamport 80] Leslie Lamport, 'Sometime' is sometimes 'not never': a tutorial on the temporal logic of programs. *Proc. of the Seventh Annual Symposium on Principles of Programming Languages*, pp.174-185, ACM SIGACT-SIGPLAN, January 1980.

[Lamport 86] Leslie Lamport, On interprocess communication. Part I: Basic formalism, *Distributed Computing*, Vol. 1, No. 1, 1986, pp.77-85.

[Lehman 85] M.M. Lehman, Approach to a Disciplined Development Process: the ISTAR Integrated Project Support Environment, *ACM SIGSOFT Software Engineering Notes*, Vol. 11, No. 4, 1986, pp. 46-60.

[Lehman 87] M. M. Lehman, Process Models, Process Programs, Programming Support, *Proceedings of the IEEE 9th International Conference on Software Engineering*, 1987, pp. 14-16.

[Lehman 89] M. M. Lehman, Some Reservations on Software Process Programming, *ACM SIGSOFT Software Engineering Notes*, Vol. 14, No. 4, June 1989, pp. 111-115.

[Levary and Lin 91] Reuven R. Levary and Chi Y. Lin, Modelling the Software Development Process Using an Expert Simulation System Having Fuzzy Logic, *Software Practice and Experience*, Vol. 21, No. 2, February 1991, pp. 132-148.

[Lin 88] Huai-An Lin, A methodology for constructing communication protocols with multiple concurrent functions, *Distributed Computing*, Vol. 3, No. 1, 1988, pp. 23-40.

[Lind 87] Mary R. Lind, A Model of Organizational Communications, *Data Base*, Vol. 18, No. 3, pp. 4-12.

[Linn 86] Richard J. Linn, The Features and Facilities of Estelle, *Protocol Specification, Testing, and Verification, V*, IFIP, 1986, pp. 271-291.

[Liskov and Guttag 86] Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development*, The MIT Press, 1986.

[Liu and Horowitz 89] L. Liu and E. Horowitz, A Formal Model for Software Project Management, *IEEE Trans. on Software Engineering*, Vol. 15, No.10 (October 1989).

[Lombardo 86] Alfio Lombardo, On the ESTELLE specification of OSI protocols, *Proceedings of the Computer Networking Symposium*, Washington, November 1986.

[Lyytinen 85] Kalle Lyytinen, Implications of Theories of Language for Information Systems, *MIS Quarterly*, March 1985, pp. 61-73.

[Macro 90] Allen Macro, *Software Engineering. Concepts and Management*, Prentice Hall, 1990.

[MacLean 89] Roy MacLean, A Functional Paradigm for Software Development, *Software Engineering Notes*, Vol. 14, No.4, June 1989, pp. 113-115.

[Martin and Fuerst 84] Merle P. Martin and William Fuerst, Communication Framework for Systems Design, *Journal of Systems Management*, Vol. 35, No. 3, (March 1984), pp. 18-25.

[Martin and Tsai 90] Johnny Martin and W. T. Tsai, N-Fold Inspection: A Requirements Analysis Technique, *CACM* Vol. 33, No. 2 (February 1990), pp. 225-232.

[Martin 91] Merle P. Martin, *Analysis and Design of Business Inoformation Systems*, Macmillan Publishing Company, 1991.

[McCracken and Jackson 82] Daniel McCracken and Michael A. Jackson, Life Cycle Concept Considered Harmful, *Software Engineering Notes*, Vol. 7, No. 2, April 1982, pp. 29-32.

[McCue 78] Gerald M. McCue, IBM's Santa Teresa Laboratory – Architectural Design for Program Development, *IBM Systems Journal*, Vol.7, No. 1, pp.2-25.

[Merlin 75] P.Merlin, A Methodology for the Design and Implementation of Communication Protocols, *IEEE Trans. on Communications*, Vol. COM-24, Number 6 (June 1976), pp.614-621.

[Merlin76] P.M. Merlin, D.J. Farber, Recoverability of Communication Protocols - Implication of a Theoretical Study, *IEEE Trans. on Communications*, Vol. COM-24, Number 9, (September 1976), pp. 1036-1043.

[Merlin 79] P.M. Merlin, Specification and Validation of Protocols, *IEEE Trans. on Communication*, Vol. COM-27, No.11, (November 1979), pp.1671-1680.

[Meyer 88] Bertrand Meyer, *Object-oriented Software Construction*, Prentice-Hall, 1988.

[Milner 80] Robert Milner, *A Calculus of Communication Systems*, LNCS 92, Springer Verlag, 1980.

[Montazemi 88] Ali Reza Montazemi, Factors Affecting Information Satisfaction in the Context of the Small Business Environment, *MIS Quarterly*, Vol. 12, No. 2, June 1988, pp. 259-276.

[Nakagawa and Futatsugi 90] Nakagawa and Futatsugi, Software Process a la Algebra: OBJ for OBJ, *Proceedings of the 12th International Conference on Software Engineering*, March 1990, pp. 12-23.

[Naur 85] Peter Naur, Programming as Theory Building, *Microprocessing and Microprogramming*, Vol. 15, (1985), North-Holland, pp. 253-261.

[NBS Report 87] *User Guide for the NBS Prototype Compiler for Estelle*, National Bureau of Standards, Report No. ICST/SNA-87/3.

[Neumann 91] Peter G. Neumann (ed.), Illustrative RISKS to the Public in the Use of Computer Systems and Related Technology, *ACM Software Engineering Notes*, Vol. 16, No. 1, pp. 10-24.

[Notkin 89] David Notkin, Applying Software Process Models to the Full LifeCycle is Premature, *ACM SIGSOFT Software Engineering Notes*, Vol. 14, No. 4, (June 1989), pp. 116-117.

[Ohki and Ochimizu 89] Atsuo Ohki and Koichiro Ochimizu, Process Programming with Prolog, *Software Engineering Notes*, Vol. 14, No. 4, June 1989, pp. 118-121.

[Olson and Bly 91] Margrethe H. Olson and Sara A. Bly, The Portland Experience: a report on a distributed research group, *International Journal on Man-Machine Studies*, Vol. 34, No. 2, February 1991, pp. 211-228.

[Osterweil 87] Leon Osterweil, Software Processes Are Software Too, *IEEE 9th International Conference on Software Engineering*, 1987, pp. 2-13.

[Palmer and Sabnani 86] J.W. Palmer and Krishan Sabnani, A Survey of Protocol Verification Techniques, *Proc. MILCOM'86 IEEE Military Communications Conference*, Monterey, California, October 1986, Vol.1.

[Pattavina 84] A. Pattavina and S. Trigila, Combined Use of Finite-State Machines and Petri Nets for Modelling Communication Processes, *Electronics Letters*, Vol. 20, No.22, October 1984, pp.915-916.

[Peterson 81] James Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[Phillips 89] Richard W. Phillips, State Change Architecture: A Protocol for Executable Process Models, *ACM SIGSOFT Software Engineering Notes*, Vol. 14, No. 4, (June 1989), pp. 129-132.

[Powers et al. 90] Michael J. Powers, Paul H. Cheney, Galen Crow, *Structured Systems Development*, boyd & fraser, second edition, 1990.

[Prieto-Diaz 90] Ruben Prieto-Diaz, Domain Analysis: An Introduction, *Software Engineering Notes*, Vol. 15, No. 2, April 1990, pp. 47-54.

[Ramamoorthy 81] C. V. Ramamoorthy, Application of a methodology for the development and validation of reliable process control software, *IEEE Trans. on Software Engineering*, Vol. 7, No. 6 (November 1981), pp. 537-555.

[Redwine and Riddle 89] Samuel T. Redwine Jr. and William E. Riddle, Software Reuse Processes, *Software Engineering Notes*, Vol. 14, No. 4, June 1989, pp. 133-135.

[Rettig 90] Marc Rettig, Software Teams, *CACM*, Vol. 33, No. 10 (October 1990), pp. 23-27.

[Roberts 89] Clive Roberts, Describing and Acting Process Models with PML, *ACM SIGSOFT Software Engineering Notes*, Vol. 14, No. 4, (June 1989), pp. 136-141.

[Robinson 90] William N. Robinson, Negotiation Behavior During Requirement Specification, *Proceedings of 12th IEEE International Conference on Software Engineering*, March 1990, pp. 268-276.

[Ross 87] M.Ross and R. van der Heever A Critical Evaluation of the Estelle Formal Description Technique in the Specification of the Message Handling System Protocols, *Proc. of the IFIP TC 6 First Iberian Conference on Data Communication*,IFIP, 1987.

[Rudin 85a] Harry Rudin, Time in Formal Protocol Specifications, *Kommunikation in Verteilten Systemen I*, Heger, D. et al. (Eds.), Berlin: Springer-Verlag, 1985.

[Rudin 85b] Harry Rudin, An Informal Overview of Formal Protocol Specification, *IEEE Communications Magazine*, Vol.23, No.3, March 1985, pp.46-52.

[Rudin 87] Harry Rudin, The Dimension of Time in Protocol Specification, *Networking in Open Systems, Proceedings 1986*, Lecture Notes in Computer Science, Vol. 248, Springer-Verlag, Edited by Gunter Muller and Robert P. Blanc, 1987, pp.360-372.

[Rumbaugh et al. 91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object Oriented Modeling and Design*, Prentice-Hall, 1991.

[Salaway 87] Gail Salaway, An Organizational Learning Approach to Information Systems Development, *MIS Quarterly*, Vol. 11, No. 2, June 1987, pp. 245-264.

[Saunders 91] Paul R. Saunders, Effective Interviewing Tips For Information Systems Professionals, *MIS Quartely*, Vol. 42, No.3, March 1991, pp. 28-31.

[Shankar 82] A. Udaya Shankar and Simon S. Lam On Time-Dependent Communication Protocols and Their Projections, *Protocol Specification, Testing, and Verification*, IFIP, 1982.

[Schwarz 82] Richard L. Schwartz and P. Michael Melliar-Smith, From State Machines to Temporal Logic: Specification Methods for Protocol Standards, *IEEE Trans. on Communications*, Vol. COM-30, No.12, December 1982.

[Senn 89] James A. Senn, *Analysis and Design of Information Systems*, McGraw-Hill, Second Edition, 1989.

[Simon 82] Gerald A. Simon and David J. Kaufman, An Extended Finite State Machine Approach to Protocol Specification, *Protocol Specification, Testing, and Verification*, IFIP 1982.

[Singer 87] Larry M. Singer, *Written Communications for MIS/DP Professionals*, Macmillian, Inc., 1987.

[Sommerville 84] Ian Sommerville, Are we Really Software Engineers? *Proceedings of the Software Process Workshop*, Surrey, UK, February 1984, pp. 59-64.

[Sommerville 89] Ian Sommerville, *Software Engineering*, Addison-Wesley, Third Edition, 1989.

[Sorenson 88a] Paul G. Sorenson, First Generation CASE Tools: All Form but Little Substance?, *Proceedings of CIPS'88 Conference*, Edmonton, 1988, pp.264-271.

[Sorenson 88b] Paul G. Sorenson, The Metaview System for Many Specification Environments, *IEEE Software*, March 1988, pp. 30-38.

[Stubbs 83] Michael Stubbs, *Discourse Analysis, The Sociolinguistic Analysis of Natural Language*, The University of Chcago Press, 1983.

[Sullivan 88] Sarah L. Sullivan, How Much Time Do Software Professionals Spend Communicating, *Computer Personnel*, Vol. 11, No. 4, September 1988, pp. 2-5.

[Swartout and Balzer 82] William Swartout and Robert Balzer, On the Inevitable Intertwining of Specification and Implementation, *CACM*, Vol. 25, No. 7 (July 1982).

[Tanenbaum 88] Andrew S. Tanenbaum, *Computer Networks*, Second Edition, Prentice-Hall, 1988.

[Tracz 79] William Tracz, Programming and the Human Thought Process, *Software-Practice and Experience*, Vol. 9, (1979), pp. 127-137.

[Tully 84] Colin J. Tully, System Development Models, *Proceedings of Software Process Workshop*, Surrey, UK, February 1984, pp. 37-46.

[Turner 89] Kenneth J. Turner, A LOTOS-Based Development Strategy, *Proceedings of the 2nd International Conference on Formal Description Techniques. FORTE'89*, Vancouver, December 1989, pp. 157-174.

[Umstot 87] Denis Umstot, *Understanding Organizational Behaviour*, West Publishing Company, Second Edition, 1987.

[Valacich et al. 91] Joseph Valacich, Alan Dennis, and J. F. Nunamaker, Jr, Electronic meeting support: the GroupSystems Concept, *International Journal on Man-Machine Studies*, Vol. 34, No. 2, February 1991, pp. 261-282.

[Vuong 87] Son T. Vuong and Allen C. Lau, A Semi-Automatic Approach to Protocol Implementation — The ISO Class 2 Trasport Protocol as an Example, *IEEE INFOCOM'87*, 1987.

[Wand and Weber 89] Yair Wand and Ron Weber, An Ontological Evaluation of Systems Analysis and Design Methods, *Information System Concepts: An Indepth Analysis*, Elsevier Science Publishers, (North-Holland), 1989, pp. 79-107.

[Ward 86] Paul T. Ward, The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing, *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 2, (February 1986), pp. 198-210.

[Weedman 91] Judith Weedman, Task and non-task functions of a computer conference used in professional education: a measure of flexibility, *International Journal of Man-Machine Studies*, Vol. 34, No. 2, February 1991, pp. 303-318.

[Weinberg 71] G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrad Reinhold, New York, 1971.

[Weinberg 83] Jerry Weinberg, A Column by Jerry Weinberg, *Infosystems*, August 1983, p.49.

[Williams 88] Lloyd G. Williams, Software Process Modelling: A Behavioral Approach, *Proceedings of the 10th International Conference on Software Engineering*, Singapore, April 1988, pp. 174-186.

[Whitten et al. 89] J. Whitten, L. Bentley, and V. Barlow, *Systems Analysis and Design Methods*, Irwin 1989.

[Wood 90] William G. Wood, Application of Formal Methods to System and Software Specification, *Proceedings of the ACM SIGSOFT. International Workshop on Formal Methods in Software Development*, May 1990, pp. 144-146.

[Yemini 84] Yechiam Yemini and James F. Kurose, Can Current Protocol Verification Techiques Guarantee Correctness? *Computer Networks*, Vol. 6, (1982), pp.377-381.

[Yourdon 89] Edward Yourdon, *Modern Structured Analysis*, Yourdon Press, Prentice-Hall, 1989.